UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

SAP AG

WALLDORF - GERMANY

TELECOMMUNICATION ENGINEERING

FINAL PROJECT

STORAGE DESIGN FOR AN AGED IN MEMORY STORE FOR UPDATES AND SIMPLE DISC BASED OPERATIONS

Author: Andrés Moreno Martínez
Tutor: Andrés Marín Lopez

21 de febrero de 2013

"One more word about computations. Although we ask readers to do a lot of computing, we don't provide very many formulas in this chapter, or throughout the book for that matter. This is deliberate. It helps us stress two things:

- *There are few "classical situations" for which specific formulas will always be applicable. Each new problem calls for examining the factors involved, considering which are most important, and basing our computation on the interactions among these factors.*
- *We believe students learn to understand concepts much better when they have to invent their own formulas. In real life, back-of-the-envelope problem solving is much more common, and much more likely to give the kinds of insights that are sought, than the use of formulas from a textbook, even our textbook."*

Michael J. Folk[1]

Acknowledgments

I would like to thank my colleges Lorena, Joaquin, Jorge, Andrea and Oscar for their company and support along the development of this project, for the incredible experience lived along these 6 months and for all the good moments shared.

I thank Alexander and Daniel their compromise with my thesis, and all the accurate and effective guidance received. Your help has determined in many stages the direction of the project. I am really thankful for all the meetings and all the time you have dedicated to improve this project.

I thank my tutor Andrés Marín, because he has supported the different orientation my final project has had and he has always provided me advise and guidance when requested.

I thank Pepe for all the feedback and the wise advises received all along my stay in SAP, every meeting has been fairly worth and I keep with me all the knowledge transferred.

I have to thank all my friends in the university. We have shared many feelings along these years and the experience was unique an unrepeatable.

I thank the association: Board of European Students on Technology, BEST, because they have given to me many incredible experiences along my studies and the opportunity to growth in many facets.

I have to thank my full family as we have walked this entire path together. I have to thank Gabriel, my brother, how he has helped me to see the reality with and open mind. I have to thank Jesus, my brother in law all his support and his friendship along these 9 years since we know each other. I have to say thank you to my sister Leonor, because she has always been there and she has always trusted in me in every single step I have done, she has given to me confidence an all her strength when I have need it. I am thankful with my niece Leonorcita because her smile and joyful has always comfort me and she has given me the opportunity to learn and see the life in a positive way every day.

And I have to thank my parents because we have gone through all the difficulties and we have challenged every exam an every year together. Thank you for giving me all the opportunities and support me in every moment of my life.

Abstract

In-memory databases have changed the database paradigm and one of the new research issues that introduce this paradigm is the non-frequently used tables available in main memory. They have to be store only in disk and the memory released. But when an isolated query is requested against these tables they have to be uploaded again in main memory. In order to avoid this time expensive and inefficient situation it is propose within this project a clever way to store the table in disk. Then when a query is requested with the help of a buffer and a reduced amount of IO operations the complete query is performed and uploading the full table in main memory is avoided.

Summary

Nowadays in-memory databases have changed the database paradigm and many new research issues appear. One of this is the non-frequently used tables available in main memory. They should be store only in disk and the memory released. But when a query is requested against these tables they have to be uploaded again in main memory.

In the old database paradigm this problem was solved thanks to the buffer manager. This engine keeps in main memory the most convenient data in relation with the demands of the user. But as this engine disappears in the in-memory databases a proper solution has to be found for the queries against the uncommonly used table. This project explores this topic.

In order to develop different solutions a prototype that simulates a table with dictionary compression is implemented. This prototype has been used as framework all along the project.

Then the problem is divided in two steps, first the procedure to store the table in disk and the second step is performing the queries using a buffer in main memory. In the second step the main bottleneck is the amount of IO operations required to complete the query.

Four approaches have been implemented along the project; each of them accomplishes different research aims. The first approach was developed to structure the solution and have a guideline available to develop more complex approaches. The second approach introduces the first optimization idea to reduce the IO operations. The third approach is the most important one presents several optimizations. And the fourth approach was developed to have a benchmark to contrast the impact of the optimizations.

One of the most important conclusions was the fragmentation of the queries procedure in three functionalities areas. Then the parameters of influence in each of the functionality areas specified. And finally the optimizations developed and tested.

The three functionality areas determined were:

- Find an element in the dictionary.
- Find a value in the index.
- Find all the attributes of a tuple.

The widest study was done in the dictionary search. A B-tree solution was designed and tested. Good results were obtained; the number of IO operations was highly reduced. Other optimizations are presented and their impact in the overall query is described; such as: ”Direct translation”, ”Sorting” and ”Conditional case”.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation of the project

The importance of the databases is growing every day as the volume of data available increases in an exponential way in different real scenarios. The data becomes more valuable when it is sorted and it can be easily access.

Nowadays is not only important to have the data organized but retrieve it in the faster way possible. Many businesses require analyzing and crossing big amounts of data, and the results obtained and the decisions take with them can be determinant for the company success.

The custom database paradigm use to have a buffer manager that keeps, in relation with the query history, a part of the data in main memory. In our time main memory has become cheaper and also there are more companies that see how the fast data analyzing and quicker response can have a high impact in the company, due to this in-memory databases have become a reality.

In-memory databases change the database paradigm and several research issues are opened. The one that justifies this project are the non-frequently used tables. These tables open the question of how worth a non-used table could be in memory, they should be stored only in disk and the memory must be released.Currently, in-memory databases queries do not deal with second storage at all, when a petition is requested against this target tables they must be fully upload to main-memory. This requires a big time consumption as upload the full table requires a big amount of IO operations. When the queries against the table become again regular this decision is worth.

In the situation where this queries do not became regular again and they are isolated requests, the time consumption required for a single query is too high and inefficient. Determining the frequency threshold when a table should be released from main memory has not been the issue of this topic.

This project works directly with the tables that have been relegated to main memory and the non-frequent queries executed against them. The full table will not be uploaded again to main memory and a buffer will be used in order to perform the query. The main bottleneck of the project is performing the query with the fewer amounts of IO operations possible. In order to achieve this each IO operation executed should retrieve the most useful information possible to avoid further ones.

1.2 Requirements

In this section requirements will be explained and also the advantages that we can take of them will be specify.

In the beginning the table is available in main memory. This table has columnstore format and dictionary compression. So we have to deal with two different situation, how the index is sotred and how the dictionary is stored. But first of all it must be analyzed what are the properties this table have. We have to take into account that the tables are aged, this means not many updates operations are supposed to be done, so, the most part of the queries will be done in order to consult data.

In this table not many updates will be perform because the table has been set as aged for this reason, and due to this is relegated to Disk. Therefore we have the advantage that we can go for a complex structure whose updates implies a big cost of time. We can go for these structures because despite much time is required to perform the updates they are quite optimal for doing searches.

The table properties can be summerized in the following ones:

- Aged table.
- Non-frequently used.
- Updates are uncommonly performed.
- Most part of the query has consult purposes.

The table has dictionary compression, then, the storage in disk can optimized following two different paths, the way we store the dictionary and the way we store the index. The dictionary compression is explained in section 3.1

- There is a big path to be runned in the Dictionary storage, one of the most important performs related with the dictionary data is searching, so we have to aim our solution in perform fast searches through the dictionary data.
- In Index files, performs take a different perspective, searches will also be needed, but we have to deal with the fact that in this data the values could be repeated, while in the dictionary is not possible. Another important matter is translating the values in the index with the current ones in the dictionary. And another point to take in to account related with this topic is the attributes retrieving, once we have found all the values that we are searching for we need to retrieve the values of the following attribute, this depends on the kind of query, but is something quite common.

When we store it in memory we have to take into account the buffer size restriction. As we are bounded by a buffer in memory we must try to retrieve as much useful data as possible, each time we perform a reading from disk. So one of the important goals is store the table in a way that is quite interrelated with the buffer size, because when the read is performed the useful data must be sequentially stored.

The most important bottleneck is the IO operations, so, the solution must aim to avoid as much as possible them, all the previous considerations should be handled taking in to account IO operations as the core issue.

Once we have specified the most important considerations is good to analyze some common ways to perform a search, and which one could fit better with the requirements.

Another important issue is the fact that the table format chosen is columstore. In all the approaches this format has been kept. There are many differences between the columstore format and the rowstore format, as they have been explained in chapter 2, but within this project the goal was not compare one more time these two solutions. Queries will be quite affected by the selection of the format of the table, but this kind of research has been already performed, and no further conclusions will be obtained taking in to account the project time boundaries. Therefore, all the approaches implemented have kept the columstore format and there isn't any optimization linked to the columnstore format or the Rowstore one. So, all the optimizations could be applied in the RowStore in the same way. It is more logical compare different approaches keeping always the Columnstore format, so, It could be compared the different solutions through the same structure.

1.3 Objectives

The main objectives of this project are three.

- Design and implementation of a prototype to work with that simulates the table structure that is available in the real product, in order to focus in the storage of it and skip the complexity of other issues not related with the scope of this project.
- Design and implementation of a group of queries which interact directly with the disk making use of a limited buffer. Store the data in the disk in a smart way in order to get good performance when running the queries. Study the procedures of the queries implemented and determine the functionalities areas that can be improved in relation with the disk storage.
- Design and implementation of optimizations in the different functionalities areas determined. Take in to account that the main bottleneck is the IO operations.

1.4 Use cases

At this point, main basis of the project has already been settled. So, it is interesting to analyze deeply in which contexts this solution could be useful. The issue here presented is a table that will not be available in memory anymore and it would be saved in disk in a smart way, in order to do later the queries against it.

Once we have taken the decision to store a table on disk, a big amount of memory will be immediately release, accordingly to the size of that table. But another inconvenient will appear, from now when we need to retrieve some data from the table more time will be required, despite in this project it has been tried to perform this in the best way, it could never compete versus the queries against main memory, as the access time to disk is quite high. These notions are summarized in figure 1.1.

	Memory requirements	Query performance
Table in disk	A small buffer in memory will be needed to perform the queries.	I\O operations against the disk are needed to retrieve the data, so, the performance will be poorer.
Table in memory	The whole table will be available in memory, which means big memory requirements will be needed.	As the whole table is in memory query performance will be quite good and fast.

Figure 1.1: Memory in disk and in memory, advantages and disadvantages

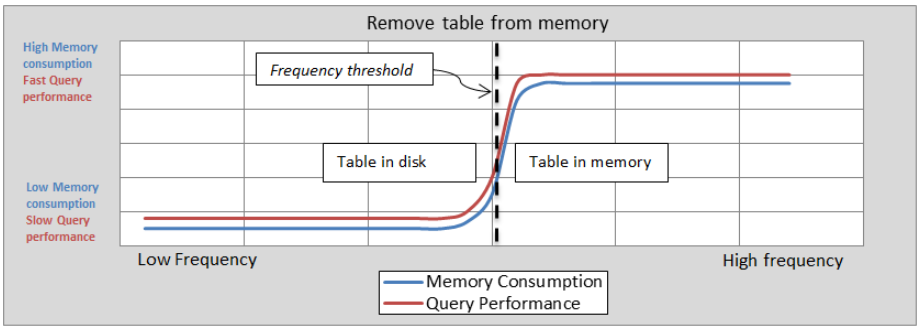


Figure 1.2: Remove table from memory, frequency threshold

When main points are well understood, we have to determine which the target tables to be stored in disk are. The main condition to take the decision to relegate a table to disk should be the frequency that we access the table. So, when a table is not commonly access anymore, which means, its data in not usually required it could be remove from memory and store it in disk. So we have to set which is the frequency when we could determine the table is not worthy anymore in memory. So there is a frequency threshold that determines if the table is not worthy anymore in memory. This situation is summarized in figure 1.2.

1.5 Content of the thesis

The memory is structured in the following disposition.

- Chapter 2 presents the importance of the databases nowadays. Then a briefly history of the databases is related until it arrives to the in-memory databases. Another issue is discussed in this chapter that has a big importance for the full project, the different ways to store the data in a file and retrieve it. These algorithms are the start point of the approaches finally implemented.
- Chapter 3 describes the prototype implemented in order to obtain a framework to work with. The table design and characteristics are described. Also the main classes of the project are introduced and how they interact.
- Chapter 4 contains the biggest section of the project, the four approaches implementation are explained. Every approach is divided in two subsections, DiskLayerManager and QuerryManager. The key points of each

subsection are described. For the most important algorithms a diagram and an example is detailed.

- Chapter 5 encloses all the tests that have been performed. Every approach has one or two experiments. The motivation of every experiment is described; then, a full study of the experiment is detailed, its results and diagrams. Finally a briefly conclusion of each result obtained is discussed.

- Chapter 6 describes what are most important global conclusions obtained in the project and what are the most useful results obtained. This chapter is divided in three mayor sections, "Fragmentation of the problem", "Parameters of influence" and "Optimizations implemented". Each of these sections describes the three most important findings in a sequential path how they lead to the following one until the optimizations are designed.

- Chapter 7 opens new questions concern to the project developed. Along the project new issues appeared but one path of improvement was followed. All the remaining issues not studied in the project are described in this chapter.

- Chapter 8 provides the code of the most important and complex method designed. Three of them are directly related with the B-tree, how to calculate the proper B-tree order for the elements, how to store it on dis, and how to perform the search on it.

Chapter 2

State of the art

In order to understand the context in which this thesis is developed, the situations when a database is required and the problems that it tries to solve must be explained. In this chapter the database concept and its history will be briefly described.

2.1 Databases and the Human Being

Databases are able to manage big amounts of data, so first of all we have to identify in which situations manipulating big amount of data is valued. In order to figure out which are these situations is good to consult the summary of "The Lowell Database Research Self Assessment"[2]. In this meeting, senior database researches gather to assess the state of database research and to recommend problems and problem areas that deserve additional focus, but first of all they determine what the main uses of database are nowadays and where they are aiming.

In the meeting, they determined that information storage, organization, management, and access, are driven by new applications, technology trends, new synergies with related fields, and innovation within the field itself. The meeting has been concluded that the nature of the sources of information are changing, how the Internet, the Web, science, and eComerce are enormous sources of information and information-processing demands. It was predicted that another big source will be the cheap microsensor technology and it has been said that the world of sensor-information processing will raise many of the most interesting database issues in a new environment, with a new set of constrains and opportunities.

The area of applications was also debated, it was determined that Internet is currently the main driving force, particularly because it has the capability of enable "cross enterprise" applications. Applications used to be intra-enterprise and could be specified and optimized entirely within one administrative domain. Now enterprises are interested in interacting with their suppliers and customers in order to provide better customer support. Another application area that is growing is the sciences, remarkably the physical sciences, biological sciences, health sciences, and engineering. These fields generates large and complex amount of data that need advanced databases.

Now some current examples where the amount of data is huge must be named. Routinely, corporate databases store terabytes (1012 bytes). But there are many databases that store petabytes (1015 bytes) of data and serve it all to users. Some important examples:

- Satellites transmit petabytes of data for storage in specialized systems.
- Storing pictures typically needs big amount of data. Repositories such as Flickr store millions of pictures and support search of those pictures. Even a database like Amazon's has millions of pictures of products to serve.
- Storing videos needs even more space, and we can fine sites such as YouTube hold hundreds of thousands, or millions, of movies and make them available.

2.2 Database Concept

"A database is a collection of interrelated data items that are managed as a single unit. This definition is deliberately broad because there is so much variety across the various software vendors that provide database systems."[3] We can find another extended definition such as a database is a collection of physical files that are managed by an instance of a database software product, in such case; a file is a collection of related records that are stored as a single unit by an operating system and an instance is a copy of the database software running in memory.[3]

"Database Management System" concept, DBMS, is the software provided by the database vendor. All the services required to manage and use the database are provided, the book available described in [3] writted by Andy Oppel specify the following:

- Moving data to and from the Physical data files as needed.
- Managing concurrent data access by multiple users.
- Managing transactions so that each of them is an all-or-nothing unit of work.
- Support for a query language.
- Provisions for backing up the database and recovering from failures.
- Security mechanism to prevent unauthorized data access and modification.

2.3 Databases history

This setion has written using the information available in [4] In the late 1960's the first commercial database systems appeared. These systems evolved from files systems. These ones provide data storage over a long period of time and of a large amount of data. However these file systems do not generally guarantee that data cannot be lost if not backed up, and they do not support efficient access to data items whose location in a particular file is not known.

File systems do not directly support a query language for the data file. Their support for a schema for the data is limited to the creation of directory structures for files. Durability is not always support, and data loses may happens if it is not backed up. Control access to data from many users at once, without allowing unexpected interactions among users (isolation) and without actions on the data to be performed partially but not completely (atomicity) is not supported neither. While they allow concurrent access to files by several users or processes, a file system generally will not modifying the same file about the same time, so changes made by one user fail to appear in the file.

The first importance applications of DBMS's were ones where data was composed of many small items, and many queries or modifications were made. Examples of these applications are:

- Banking systems: maintaining accounts and making sure that system failures do not cause money to disappear.

- Airline reservation systems: these, like banking systems, require assurance that data will not be lost, and they must accept very large volumes of small action by customers.

- Corporate record keeping: employment and tax records, and a great variety of other types of information, much of it critical.

2.3.1 Relational DB

Following a famous paper written by Ted Codd in 1970, database systems changed significantly. Codd proposed that database systems should present the user with a view of data organized as tables called relations. Behind the scenes, there might be a complex data structure that allowed rapid response to a variety of queries. But, unlike the programmers for earlier databases systems, the programmer of a relational system would not be concerned with the storage structure. Queries could be expressed in a very high-level language, which greatly increased the efficiency of database programmers.

By 1990, relational database systems were the norm. Yet the database field continues to evolve, and new issues and approaches to the management of data surface regularly.

2.3.2 HANA, the new paradigm

"SAP in-memory database means a change in the database paradigm. HANA keeps the primary copy of its data in-memory to provide up-to-date data for fast ad-hoc processing in-memory at any time" [5]. This means the "Buffer Management" deals with data in a different way, as it is not needed anymore decide which must be the data available in memory, as it is all there, so the optimizations thought for this model are not any more useful, and a new world of possibilities and optimizations is open in this new paradigm.

Figure 1 present the different parts that enclose the classical database system. The parts described as "Index/File/Record manager", "Buffer manager", "Buffers" are not available anymore. Storage will only be used for backup issues.

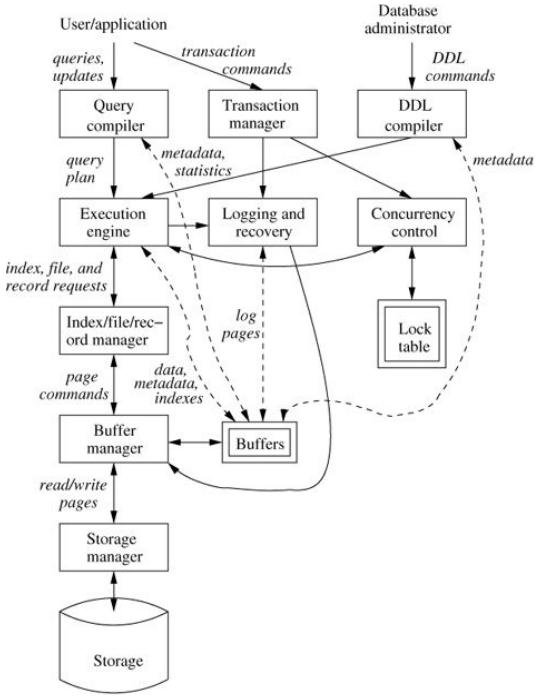


Figure 2.1: Hana changes the paradigm - Image obtained through [4]

2.4 File structures

As a result of the new paradigm open by HANA database, new optimizations are required. In the previous paradigm we found that aged tables that were not anymore use, or not used for a long time don't require any attention. When these tables are used again "Buffer Manager" will decide whether or not to maintain part of these tables in the buffer, accordingly to the manager behavior. So, aged tables cases where already solved within the previous paradigm.

But now the DBMS works in a different way and doesn't deal anymore with second storage as all the data is available in main memory. So the situation that presents this new paradigm is that aged tables that are barely use should be store in the second storage. If they are somehow needed again for doing some small interactions, these aged tables need to be uploaded again in main memory.

It would be good to avoid the time costs of uploading the full table back in memory, in situations where a small amount of queries will be do against these aged tables, that is the common case, and then forget them again. So it could be interesting to do these queries directly against the second storage, despite of retrieving the whole table back in memory.

2.4.1 File structures access

This section was writted using the ideas and the algorithms presente in [6]

File data has to be oranized in such a way that retrieve it must be quick.

We have to be aware of the different ways to store the records in a file; we have the following structures available:

- Fix the Length of Fields
- Begin Each Field with a Length Indicator
- Separate the Fields with Delimiters
- Keyword=Value

Also we have to take in to account the different ways to perform a search in a file. The cost of accessing secondary storage is huge. Then the main goal is minimize the number of disk accesses and therefore minimize the amount of time expended. The most common algorithm are described in the following list and their characteristics related with the project issue are summarized in figure 2.2

- Sequential search- this method implies retrieve data until the value is found. Then in the worst case the full file must be read in order to find the value.
- Binary search- this method avoids the need of reading the full file. But element sorting must be done first in order to be able to jump to the middle value of the list, recursively until we found the element required. It is also need that we have the capability of know where the middle position is. Despite we dispose everything in order to be able to perform a binary search is not a good solution. The binary search implies that only one element of the IO operation done will be useful, and then we have to do another IO operation.
- Keysorting- this methods take the advantage of sorting the records. But only the record Key are needed to be sorted. Later the full file is not needed to be read but only their sorted record keys. The main disadvantage of this method is computations cost of the sorting. It will be interesting when the file is hardly modified. This idea is close to the dictionary compression explained in section .

Keysorting introduces a modification in the order of the elements in order to improve the search performance but it always keeps track of the original order to retrieve the correct values. Keysorting can be used as baseline idea and go further. Another possibility is storing the elements in a B-tree disposition in order to reduce the number of IO operations to perform the search. As it was especified in Keysorting method the main disadvantage is the computation cost of the B-tree creation. It will also be worth when the table is hardly modify that it is just the scenario that was presented in section 1.2.

Search algorithm	Characterisics related with performing the search through a buffer
Sequential search	In the worst case the full dictionary has to be read. Many amount of IO operations are needed.
Binary search	Only one element will be useful when the buffer is filled, because of the way the binary search works, that element will determine the next jump.
B-tree search	The main advantage of the B-tree search is that the roots can be stored sequentially, and the elements for each node also. Then all the elements that fill the buffer are useful. The search can be performed with a few amount of IO operations.

Figure 2.2: Search algorithm comparisson

Chapter 3

Architecture

3.1 Table characteristics

The table used as baseline for the project implements dictionary compression. This means every column is represented by an index and a dictionary. The dictionary contents all the different values stored in the column, but it does not have repeated values. The index is composed by pointers to the position of their real values in the dictionary. An example is described in figure 3.1

3.2 Prototype

Once the requisites are understood and analyzed and the environment selected, the next step is set up a prototype and maintain it the entire project long. For this project a three classes' solution has been designed. Each of the classes provides a certain amount of functionality. The functionality within each class is quite interrelated and the classes interact between them in a successive way, as it is going to be explained.

Order	Original Column
1	Andres
2	Leonor
3	Leonor
4	Jesus
5	Gabi
6	Laura
7	Gabi
8	Leonor
9	Andres

Dictionary		Index
1	Andres	1
2	Leonor	2
3	Jesus	2
4	Laura	3
5	Gabi	5
		4
		5
		2
		1

Figure 3.1: Table with dictionary compression

Class	Functionality
MainTableManager	Simulate Main Table (Vector of Columns with Index and dictionary)
DiskLayerManager	Simulate Disk Layer (Manage several files)
QueryManager	Simulate Query engine (Interface that provides different queries)

Figure 3.2: Main Classes

3.2.1 Main Classes

This project is composed by three main classes, before introduce them, it is important to present first ColumnI_D_s class. All these classes are briefly explained below.

- ColumnI_D_s - this class represents a column with an Index and a Dictionary, also the length of the longest string that has the column is stored.
- MainTableManager - this class represents one full table, which means a vector of ColumnI_D_s. It is remarkable the function "ImportCSV" that is able to generate a complete MainTableManager object with the Index and the Dicitionary of each column, from a csv file where just the values of the each column are stored.
- DiskLayerManager - this class is in charge of storing the MainTableManager on Disk. Each approach performs this operation in a different way. The result will be a set of file that contains the data.
- QueryManager - this class interacts with the files created by DiskLayerManager to retrieve the data specified in a query. Each approach deals with the data in the files in a different way.

The main objective of this first approach was to have all the classes implemented and working together and set the foundations of the project. In order to accomplish this goal an easy and fast approach was implemented.

It is important to be said that MainTableManager has not been modified along the whole project. However DiskLayerManager and QueryManager are quite linked, therefore, each approach has its own DiskLayerManager and QueryManager objects.

Another important class is AgedDataStorage, this one owns the main. In this class the three main objects already explained are instantiated and their functionalities are invoked, as it is going to be explained in section 3.2.2. In the first two approaches tests were also perform in this class.

The last class to be explained is TestSet. This class arises in approaches 3 and 4. In these approaches evaluation functionalities are granted to TestSet class, in order to clarify which are the different parts of the project, and facilitate the testing of the project.

3.2.2 Classes interaction

After the main classes have been presented, it is interesting to jump in the topic how they work together. In order to do this we can group the most import

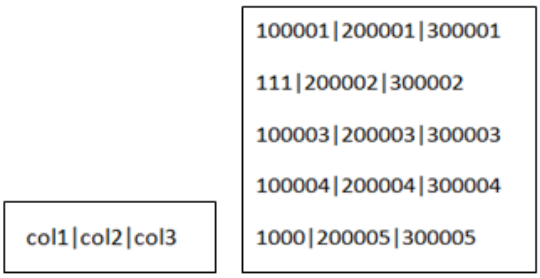


Figure 3.3: TBL and CSV files

functionalities in three steps:

Step 1

In the beginning we only have two files that have all the data available. One of them is a ”csv” file and the other is a ”tbl” file. In the ”csv” file we have information about the columns, the names of all the columns are accessible, and this information is displayed as it is shown in the left box of figure 3.3, as we can see it is store in columnstore format. In the ”tbl” file we have the actual data of each column displayed as it is shown in the rigth box of figure 3.3.

Step 2

At this point, the next action needed is storing the table again in files, but somehow, that latter when queries are done against them we obtain good performance. Good performance in relation with the requirements specify in section **??**. In order to do this we have to instanciate DiskLayerManager. In order to do this the class have one method available called: addDataToFile_aproach0X(class MainTableManager *MTMp,int buffersize); X correspond to number of the approach, MTMp must be a pointer to an instance of a MainTableManager and the buffersize is the buffer size restricction. In the approaches implemented in this project we can obtain two kinds of results, three files are generated: ”header”, ”dic” and ”index”; or only two: ”header” and ”data”. The content of these files will be deeply explained in ”Implentation” chapter.

Step 3

When step 1 are step 2 are done, objects DiskLayerManager and MainTableManager can be free from memory. They are not needed anymore as all the data is available on disk. Then, we have to create an instance of QueryManager, this class provides us several queries to get the data from disk. In figure 3.4 the whole process is summarized.

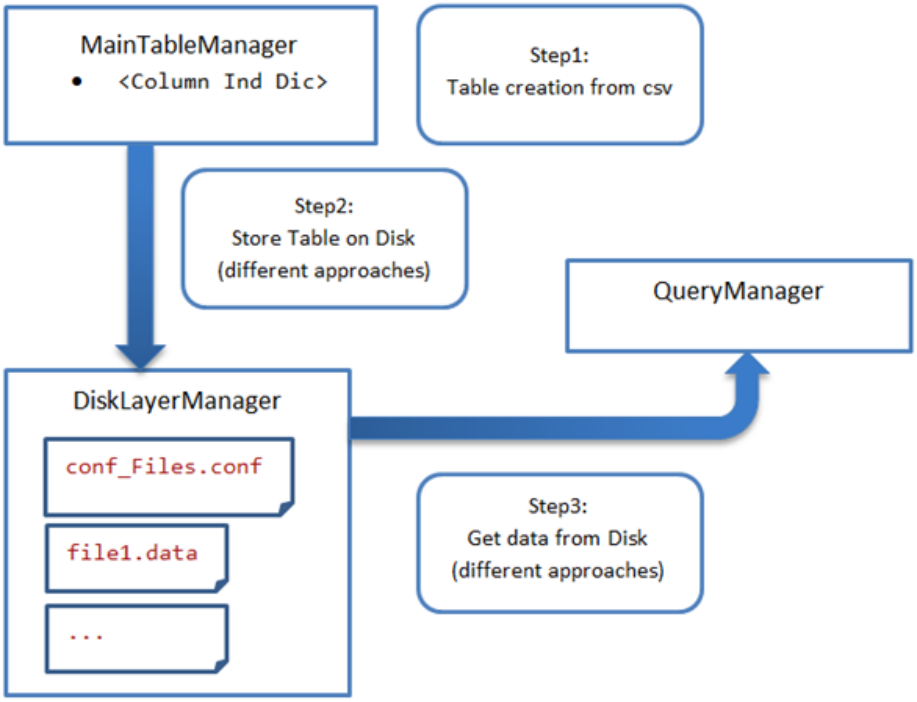


Figure 3.4: Classes interaction

Chapter 4

Implementation

In this chapter all the approaches implemented will be explained. The motivation of each approach will be presented first and how has been the evolution of the research and why the next approach has been done. Also the most important ideas of each approach will be commented and some briefly conclusions.

It is important to specify that the first two approaches use the buffering from windows. But in approaches 3 and 4 the buffering functionality is disabled and the impact of the IO operations can be studied in a most accurate way.

4.1 Environment

The operating system selected for the project has been Microsoft Windows 7. It has been chosen because the prototype of the project could be done in the environment that we select, taking care to keep it the entire project long, and doing all the tests and comparisons through the same conditions. Therefore, the mayor advantage of choosing Microsoft Windows 7 against Linux is that SAP offers IT support for Microsoft Windows 7 while no support is offer for Linux.

The framework used among the ones available has been Microsoft Visual Studio 2010, because is quite powerful. This product offers good resources for development issues, such a human friendly interface for programing or quite potent debugging tools.

The program language used is C++, as it is the one that used for the whole final product. Therefore, in case it is finally promoted to be integrated in the product it facilitates the procedure and code reutilization.

4.2 Approach 01: Uncompressed Storage

The main objective of this first approach was to have all the classes implemented and working together, set the foundations of the project and obtain a baseline. In order to accomplish this goal a simple comprenhensive solution was implemented.

Column name		Col1	Col2	Col2
Data	1	Maria	A	4
	2	Ana	B	5
	3	john	A	6
Longest element in bytes		5	1	1

Figure 4.1: Example Table

4.2.1 DiskLayerManager

The DiskLayerManager object store each column of the table in a quite non-complex way. It just undoes the dictionary compression and stores each column with their real values. The main goal of this approach was to code the approach fast and wee how them all interacts. Another goal was to have an easy approach to have a benchmark to compare with more smart solutions, and then be able to determine how better the most complex approach are and how worthy they really are.

The table is stored in two files ".data" and ".header". The header file contains information about each column, the name of the column and the length of the longest element; this is written in the first line. In the second line the number of rows of the table is written. An example of a header file is shown in figure 4.2.

The data file contains the values of each column arranged in columnstore format. Each value is stored in such a way that it occupies in memory as much bytes as the longest. If one element is smaller padding is added. This is done to be able to jump easily to the value and the column that we want, and preserve the main goal of this approach: keep it simple. An example of a data file is shown in figure 4.3.

4.2.2 QueryManager

The QueryManager object in this first approach has only one query available. This first approach could have been a good benchmark approach, but it was dismiss and no more queries were implemented. This was because it was discovered in following stages in the project that this approach had enabled the caching done by Windows and it do not let evaluate well the speed of the approach. Therefore, I did not come back to this approach to extend it and compare it with other approaches. The query implemented was:

"SELECT * FROM tableName WHERE columnName = elementName"

This query is performed in three phases:

1. Find out in which position starts our column - using the header file
2. Read the data and find the coincidences - using the data file
3. Read the information in the other columns of that row - using the data file

In order ot summerize how this approach works, and example is shown. This example stores table presented in figure 4.1 in file described in figures 4.2 and 4.3; finally an example of a query will be specified.

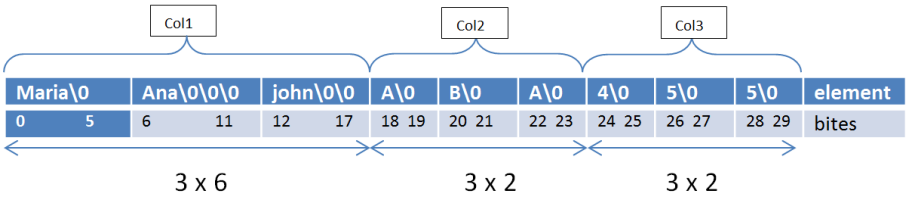


Figure 4.2: Example .data file

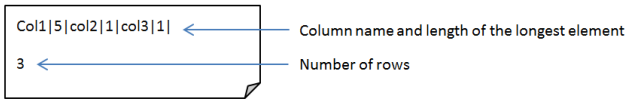


Figure 4.3: Example .header file

Once the table is stored in disk, we are going to specify how a query works, in this example the query executed will be: "SELECT * FROM table1 WHERE Col2="B";" and "(Buffer size = 5)"

1. Find out in which position starts our column - using the header file. Col2, so col1 must be skipped(3 elements of 6 bytes), jump to 3 x 6 = 18.

2. Read the data and find the coincidences - using the data file. Buffer size is 5 bytes, elements in Col2 has length 2, so the maximun number of elements that we can read each time is 2, 4bytes. In this example, after read the full data file we find one coincidence in position 2.

3. Read the information in the other columns of that row - using the data file. We have to calculate where are the other values of the row found.

   - Col1 - (2-1) x 6 = 6 - Ana
   - Col2 - (3x6)(Skip Col1) + (2-1) x 6 = 20 - B
   - Col3 - (3x6)(Skip Col1) + (3x2)(Skip Col2) + (2-1) x 6 = 26 - 5

4.3 Approach 02: B-tree 1 level

Once the foundations are settled and the project well structured, next jump must be done, a more sophisticated solution must be implemented. In order to do this and taking in to account the ideas explained in chapter 2 a B-tree search is going to be implemented.

In this approach the optimizations has been focused in the dictionary. Dictionary is used in two ways, to translate the values from the index and to check if a value is available on them. I have focused in optimize the searching in the dictionary. As it has been explained in chapter 2 binary search is not optimal because when an IO operation is done a small amount of useful data is retrieve. But if I manage to store it in B-tree format many advantages could be taken.

If B-tree format is selected for the dictionary the main advantage that we can obtain is the optimization of the IO operations, If we arrange the data in

the dictionary file in such a way that when an IO operation is perform all the nodes of one level of the tree are retrieved, then, all the information will be useful, as we can determine which is the next block that we have to read, this means the leaves of that node.

There is another important advantage that B-trees have; this is the amount of elements needed to be read in order to perform one search, as we jump for one level to the leaves of the node that corresponds we can skip many elements.

4.3.1 DiskLayerManager

The first idea was to implement a full B-tree, but while doing this approach a new requisite was determine, so only a 1 level B-tree was implemented, to jump a new approach that accomplish the requisite. The new requisite was to avoid Buffering from windows.

In this approach padding is also added in each element. This is done in order to simplify the implementation. In next approach padding won't be added. Because, despite the implementation is simplified, more not useful data is read in each IO operation and more disk wasted.

In order to store the dictionary elements as a 1 level B-tree an auxiliary function has been implemented. This function name is "generate1NtreeOrder" and uses the following methodology:

1. The function has only two parameters: the number of rows that the dictionary has (rowsDicAux) and the number of rows that fits in the buffer (buffersizeRows).

2. If rowsDicAux is smaller than buffersizeRows, the elements won't be rearranged.

3. If it is bigger, we calculate how many leaves will have each root node. We already know that we will have buffersizeRows number of roots.

4. Then, we save the first element of the dictionary and then we jump all the elements that will be his leaves, and we save the next root. We perform this way until all the roots are saved.

5. When all the roots are selected we save the leaves sorted, skipping the roots that are already saved.

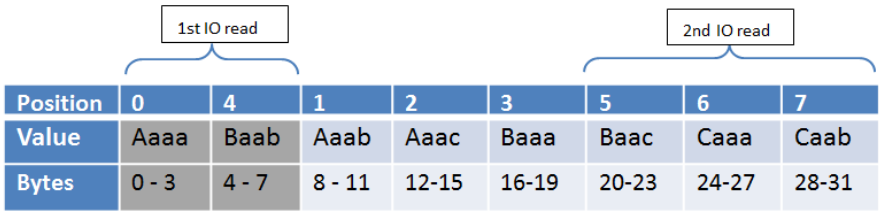
Example: We have a dictionary that has 8 elements. The buffer size is 8 bytes and max length of elements is 4 bytes so 2 elements fits in a buffer. So, we will have two roots. Then we have to determine which the roots are and which the leaves are.

1. Number of rows that the dictionary has (rowsDicAux): 8 Number of rows that fits in the buffer (buffersizeRows): 2

2. rowsDicAux is bigger than buffersizeRows, therefore the elements will be rearranged.

3. number of roots = buffersizeRows = 2. We calculate how many leaves will have each root node. 8 / 2 = 4 -> each root will have 3 leaves.

Column name		Col1
Data	0	Aaaa
	1	Aaab
	2	Aaac
	3	Baaa
	4	Baab
	5	Baac
	6	Caaa
	7	Caab
Max string length		4



Position	0	4	1	2	3	5	6	7
Value	Aaaa	Baab	Aaab	Aaac	Baaa	Baac	Caaa	Caab
Bytes	0 - 3	4 - 7	8 - 11	12-15	16-19	20-23	24-27	28-31

Figure 4.4: Example: Dictionary file construction

4. Then, we save the first element of the dictionary (element 0) and then we jump all the elements that will be his leaves (1,2,3), and we save the next root(element 4) and their leaves (5,6,7)

5. When all the roots are selected we save the leaves sorted, skipping the roots that are already saved. (1,2,3,5,6,7). This full examples is shown in figure 4.4.

In the figure 4.5 is shown how the tree of the previous example could be represented. Once the B-tree 1 level order is calculated the next step is just save the elements in the dictionary file. "generate1NtreeOrder" method is available in Annex 8.1

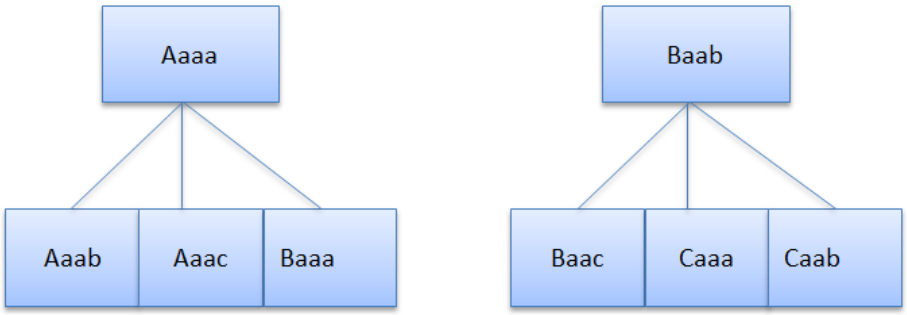


Figure 4.5: Example: Dictionary tree

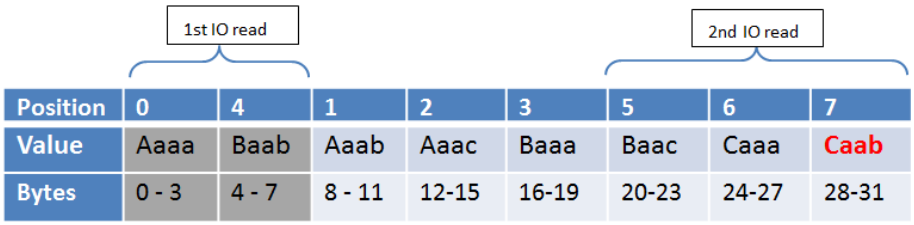


Figure 4.6: SELECT * FROM table WHERE col1 = Caab;

4.3.2 QueryManager

In this approach 4 queries have been implemented. This class uses the advantage or the 1 level N-tree to perform the queries. The queries available are:

- SELECT * FROM tableName WHERE columnName = elementName;
- SELECT columnName FROM tableName;
- SELECT * FROM tableName;
- SELECT * FROM tableName WHERE col1= elem1 AND col2= elem2;

This class provides also the auxiliary method posInDic1NtreeOrder, this method is used to get the value of a position (row) in the dictionary file. This method is needed because when we need to retrieve the value of a certain row of a certain column, the exact position where is located in the dictionary file must be first calculated.

The functionality of the method that implements the query "SELECT * FROM tableName WHERE columnName = elementName;" is presented through an example. In order to show how this approach avoids IO operations a query is executed against the table presented in figure 4.4, the query executed is "SELECT * FROM table WHERE col1 = Caab;". As it is specified in figure 4.6 only two IO operations are needed, and some leaves of the tree are skipped.

4.4 3rd approach: Multilevel with sector size requirements

This approach is the most important one of the fourth approaches implemented as it contains the most interesting algorithms designed. In this approach are placed the key points discovered in the research: a method that creates a B-tree, store directly the dictionary's position of the element in the index to do the translation faster, sorting of elements found to avoid IO operations and conditional case when dictionary fits in the buffer.

It is important to describe that this is the first approach in which the buffering from Windows Operating System is disabled. Now it can be studied better the impact of the IO operations, as not any advantage from the Operating System is used.

Following the ideas already presented in approach 2 in this approach also a B-tree implementation has been done for the dictionary in order to reduce the number of IO operations, and to perform fast searches on it.

In this approach the B-tree is fully created. In approach 2 it has been discovered that create the B-tree from top to down, is difficult. In the scenario were the project is developed we have already explained that not many inserts or updates will be done. Therefore when we create the B-tree from the table we can assume that no more elements will be added or modified. So, when we create the B-tree we start with the leaves of the last level and then we go to the roots, this methodology will be explain in detail in section DisklayerManager.

This approach has two different versions, because a modification was done in order to change the way of performing the queries. The modification was avoid the sorting of the elements found when the dictionary fits in the buffer, this is explained in detail in section QuerryManager.

4.4.1 DisklayerManager

In this approach 3 files are created in order to store the full table, these files are ".header" ".dic" ".index".

- In header file is stored the name of each column, the number of chunks needed to store the dictionary of each column and the number of elements that has the dictionary of each column, after thar, the number of rows of the table is stored.
- In dic file the dictionary of each column is stored. The dictionary is stored in a B-tree format. The B-tree is calculated with an auxiliary method called "generateMLneworder". This method will be explained in detail in this section. Its implementation is available in Annex 8.3
- In index file the index of each column will be stored. The relative position of the element in the dictionary will not be stored, but directly the position of the element in dictionary file. Thanks to this the translation from the index file to the real value in the dictionary file can be fast and we can reduce the number of IO operations. These advantages will be explained in detail in this section.

The method that creates the three files is "addDataToFile_aproach03(class MainTableManager *MTMp,int buffersize)" . This method is provided in Annex 8.2. Its procedure can be explained in the following steps:

1. Open the Dictionary file and the index file
2. For each column of the dictionary we calculate a new order for the elements that represents a B-tree. This will be explained in section 4.4.1.
    - Once the new order is calculated we store the elements in the Dictionary file and we store also some additional information, this is show in figure 4.7.
    - We save in one vector the position in the dictionary file where each element is saved.

- We store in the index file directly the position that the elements have in the Dictionary file.

3. We close the Dictionary and the index file

4. We get the name of the column and the size of the dictionaries from the table and we store this information in the header file.

Dictionary file

The elements of the dictionary are stored as a B-tree. This is done in order to avoid IO operations when we execute the queries. The most important method needed to create the B-tree is the auxiliary one: "generateMLneworder". This auxiliary method creates a B-tree with the elements of the dictionary. This B-tree is created grouping the elements in chunks. Each chunk contains the maximum number of elements that fits in a buffer. Then, each node has one chunk as leaves, and each element of the chunk is a node. The B-tree is created in a different way from Approach 2nd, in that approach the method first select the roots nodes and then the lower levels. But in this approach the last leaves are grouped in chunks first and while doing this we select the nodes for the upper level. Once we have done this, we do the same with the nodes selected for the upper level. We do the same procedure until we arrive to the roots nodes. So the method implemented is recursive. This auxiliary method return a struct named "calledNewOrderMultiLevel", this struct contains three vectors:

- vector<int> newOrderResult: New order for the elements that creates a B-tree.
- vector<int> chunkByLevel; Number of chunks that each level of the tree has
- vector<int> elementByChunk; Number of elements that each chunk has

The parameters of the method are:

- vector<string> columnString; Elements sorted in alphabetical order to be sorted as a B-tree
- vector<int> columnPosition; Alphabetical order position of elements stored in columnString
- int buffersize; Size of the buffer in bytes
- newOrderMultiLevel my_noML;

First, the method fits all the elements possible in one buffer, we named this a chunk, and the following element is reserved as a node for the upper level. This is done with all the elements of the vector ColumnString. At the end we have several chunks and some nodes reserved for the upper level, then we call again the method, in recursive way. In order to understand better the functionality of the method, a diagram is shown in figure 4.7. And an example is detailed in figures 4.8, 4.9 and 4.10.

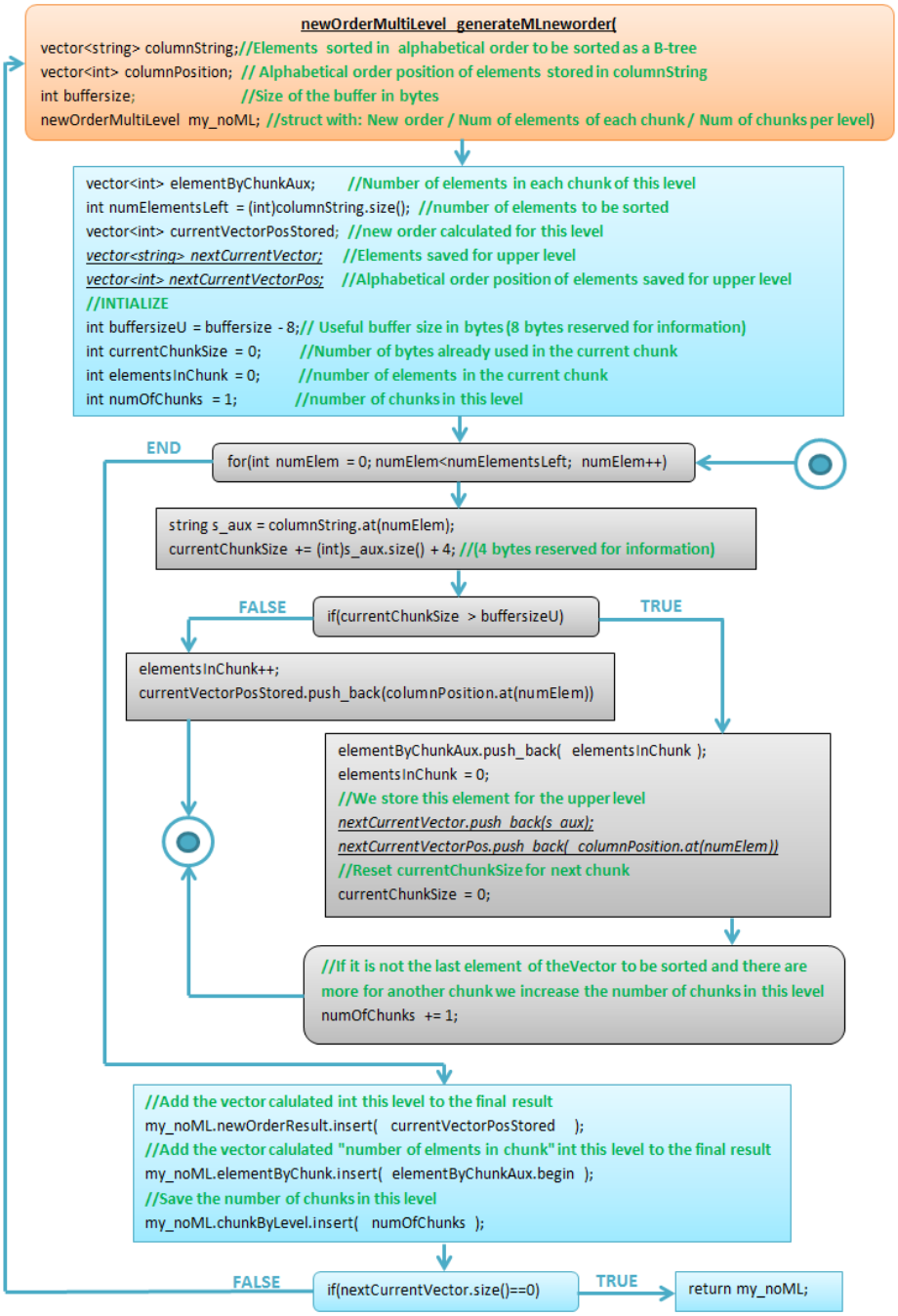


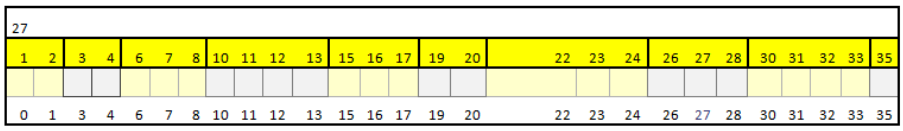
Figure 4.7: Method: generateMLneworder

char0	Char1	Char2	char3	char4	char5	char6	char7	char8	char9	Element	bytes
A	a	a	a	a	A					0	10
A	b	b	b	b	b					1	10
A	c	c	c	c	c					2	NextLoop
A	d	d	d	d	d					3	10
A	e	e	e	e	e					4	10
B										5	NextLoop
C										6	5
D										7	5
E										8	5
E	a	a	a	a	a					9	NextLoop
F										10	5
G										11	5
H										12	5
J										13	5
K										14	NextLoop
L										15	5
M										16	5
M	a	a	a	a	a					17	10
N										18	NextLoop
N	a	a	a	a	a					19	10
N	b	b	b	b	B					20	10
N	c	c	c	c	c					21	NextLoop
N	d	d	d	d	d					22	10
O										23	5
P										24	5
Q										25	NextLoop
R										26	5
R	a	a	a	a	a					27	10
S										28	5
T										29	NextLoop
U										30	5
V										31	5
W										32	5
X										33	5
Y										34	NextLoop
Y	a	a	a	a	a					35	10

Figure 4.8: Column example to be stored as a B-tree

1st Loop

CURRENTVECTOR = FULL DICTIONARY / 10 chunks and 27 elements in this level / New order calculated=

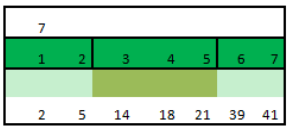


2nd Loop

CURRENTVECTOR =

char0	Char1	Char2	char3	char4	char5	char6	char7	char8	char9	Element	bytes
A	c	c	c	c	c					2	10
B										5	5
E	a	a	a	a	a					9	NextLoop
K										14	5
N										18	5
N	c	c	c	c	c					21	10
Q										25	NextLoop
T										29	5
Y										34	5

3 chunks and 7 elements in this level / New order calculated=



3rd Loop

CURRENTVECTOR =

char0	Char1	Char2	char3	char4	char5	char6	char7	char8	char9	Element	Bytes
E	a	a	a	a	a					9	10
Q										25	5

1 chunk and 2 elements in this level, roots / New order calculated=

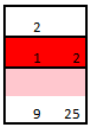


Figure 4.9: generateMLneworder example

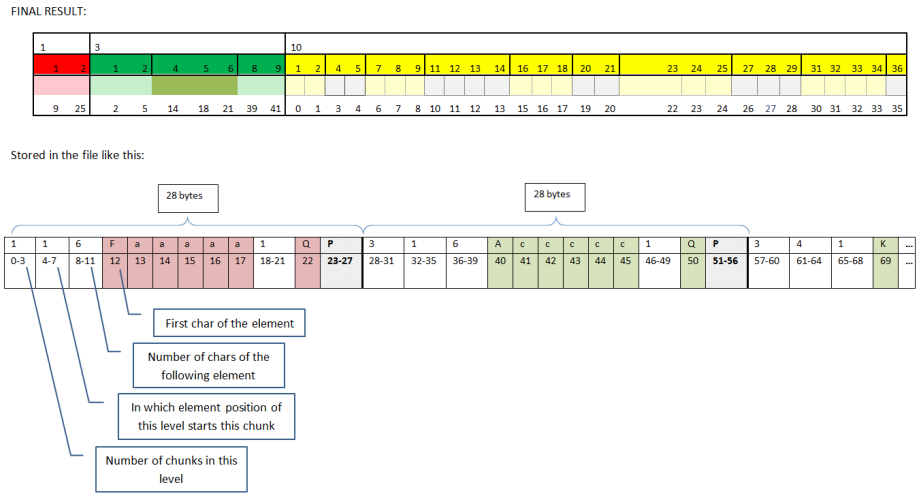


Figure 4.10: generateMLneworder result

4.4.2 QueryManager

In this approach 5 queries have been implemented, the following ones:

- SELECT * FROM tableName;
- SELECT columnName FROM tableName;
- SELECT * FROM tableName WHERE columnName = elementName;
- SELECT * FROM tableName WHERE columnName = elementName AND columnName2 = elementName2;
- SELECT columnName FROM tableName WHERE columnName = elementName;

In order to perform these queries 6 auxiliaries methods have been implemented. Three of them are used for to search data in Dictionary file and Index File. These methods are explained in detail in this section but first a brief explanation of their functionality is presented:

- findPosStringInDic: Find a string in a dictionary and return the position in the dictionary file.
- findPosValueInIndex: Find the relative positions where a value appears in the index.
- findColValueForRows:Find the values of the columns indicated for the rows specified. This method have two versions, in this project the two versions have been compared in order to check which one works better.

The other three have different functionalities:

- openFileNoBuffering: this method open a file and avoids the buffering from windows when reading it. It returns the HANDLE to manage the file.

- getColumnsInfo: this method returns and struct with information about the columns: the names, the number of chunks and the number of elements that have the dictionaries, the position where each column begin in the dictionary file and the number of rows that the table has.

- sort_indexes: this method sorts one vector of ints from lower value to higher value and return the new order.

findPosStringInDic

This method uses the advantage of the B-tree format in the dictionary file in order to search an Element. It reads first the chunk where the root nodes are and then it jumps to the leaves of the convenient node. He performs the same procedure until it founds the element asked. Thanks to the B-tree format the search can be performed with a small amount of IO operations. This procedure is explained in the following example. Example: For this example the table presented in figures 4.8,4.9 and 4.10 is used. For this example we search element "Raaaaa"(position 27 of the dicitionary).

1. In the first IO operation we read all the roots, there are two roots and then we calculate which chunk of the lower level we have to read.

   - 1st IO : chunk read [1 / 0 / 6 / *Faaaaa* / 1 / *Q* P]
   - Compare: higher than "Faaaaa" (+1) and than "Q" (+1)
   - Then: Jump to chunk 3 of Green level.

2. In the second IO operation we read the specific chunk of the lower level and we calculate again which chunk of the lower level we have to read.

   - 2nd IO : chunk read [3 / 8 / 1 / *T* / 1 / *Y*]
   - Compare: Lower than "T"
   - Then: Jump to chunk 8 of Yellow level.

3. Finally in this chunk we found the element we are looking for.

   - 3rd IO : chunk read [10 / 20 / 1 / *R*/ 6 / *Raaaaa*/ 1 / *S*]
   - Compare: higher than "R" match with "Raaaaa"
   - Return position 11x28bytes = 308; 308 + 19 = 327

The implementation of this method is available in Annex 8.4.

findPosValueInIndex

This method finds the rows where an element appears in a specific column. The search is performed in the index file, where all the values are stored as ints thanks to the dictionary compression; due to this the search could be done faster. A vector with the positions of the rows found will be returned.

findColValueForRows

This method retrieves the value of several rows for a specific column. The rows selected and the column desired are specified as parameters, also 2 handlers to manage the index and the dictionary file. The diagram presented in figure 4.10 explains the procedure of this method. An example is presented below.

It is known in which positions appears element "16" in column AGE, these positions are stored in parameter: "vector<int> rowsPos"

- Example: rows (4,17,200,352) (this values are sorted)

We want to get the values of these rows in another column, specified in parameter: "int colSelected"

- Example: column CITY (col 3)

It is important to specify that in the index file all values are stored as ints and it is not stored the relative position in the Dictionary, but, the position of the element in the Dictionary file. So we can directly get the value of the element just opening the Dictionary file and read the position specify in the index file, without any other calculation.

A read is performed in the index file to retrieve the first element, row 4 of column CITY, we get 8324 for example. But with the second element and the others we first check if it is in the buffer already read in the previous loop. It is not needed to read again if it is already available. After perform this operation with all the rows selected we obtain the vector:

- Example: postInDIC = (8324,7500,7003, 8324) (this values are not sorted)

The positions are not sorted, so they should be sorted, then, same advantage can be taken as it was did before, and not so many readings have to be done if the values are repeated or they fit in the buffer already read.

- Now: postInDIC = (7003,7500,8324,8324) Now they are sorted, the original order is also saved to return the values in the order asked, (2,1,0,0)

The main disadvantage with this approach arises when the dictionary is quite small and it fits in the buffer, in this situation there is no advantage in sorting the vector, as only one read is needed. And when the number of elements found is high the sort operation needs a lot of time. Due to this conclusion another version was done to compare how much affects this. Then there is a conditional case that when the dictionary fits in one buffer the sorting is skipped. Finally the index file is read and the values obtained:

- result_s_final = (Walldorf, Leganes, Madrid, Frankfurt)

Queries implemented

In this approach 5 queries have been implemented they are based in the auxiliary method explained above. The queries methods are a combination of the 3 main auxiliary methods:

- findPosStringInDic

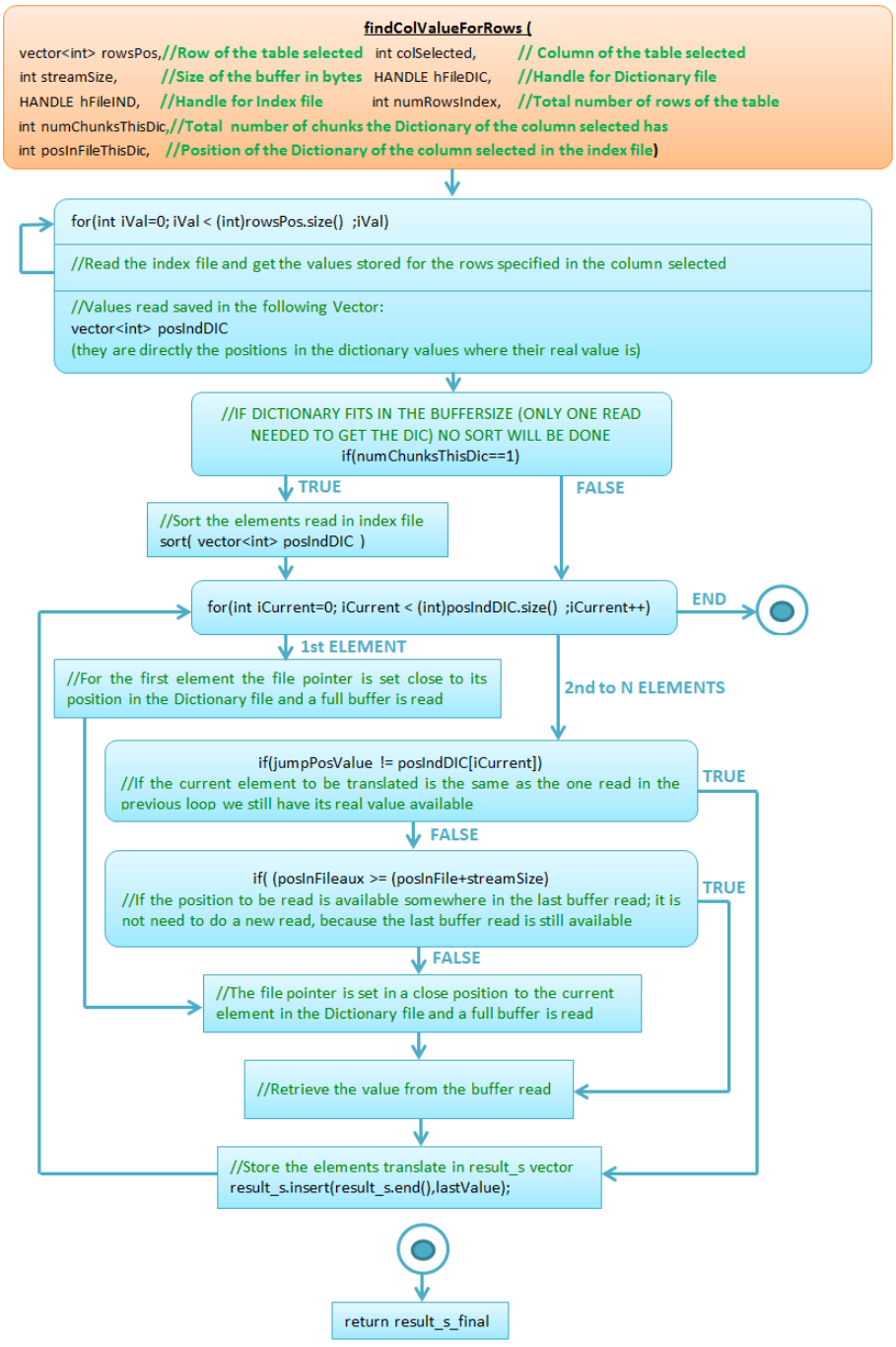


Figure 4.11: findPosValueInIndex diagram

- findPosValueInIndex
- findColValueForRows

SELECT * FROM tableName WHERE columnName = elementName AND columnName2 = elementName2;
Name:
QueryManagerRE::QueryS5_approach03
Returns:
vector<vector<string»
Parameters:
(string tableName,string columnName,string elementName,string columnName2,string elementName2,int streamSize, string colSeleted,int colSel);

Procedure:

1. Elementfound1 = findPosStringInDic(elementName , columnName)
2. Elementfound2 = findPosStringInDic(elementName , columnName2)
3. Rowsfound1 = findPosValueInIndex(Elementfound)
4. Rowsfound2 = findPosValueInIndex(Elementfound)
5. Rowsvalues = findColValueForRows(Rowsfound1 U Rowsfound2 , All columns)

SELECT * FROM tableName;
Name:
QueryManagerRE::QueryS4_approach03
Returns:
vector<vector<string»
Parameters:
(string tableName,int streamSize);

Procedure:

1. Rowsvalues = findColValueForRows(All rows , All columns)

SELECT columnSelected FROM tableName WHERE columnName = elementName;
Name:
QueryManagerRE::QueryS3_approach03
Returns:
vector<string>
Parameters:
(string tableName,string columnName,string elementName,int streamSize,string columSelected);

Procedure:

1. Elementfound1 = findPosStringInDic(elementName , columnName)
2. Rowsfound = findPosValueInIndex(Elementfound)

3. Rowsvalues = findColValueForRows(Rowsfound , columnSelected)

SELECT columnName FROM tableName;
Name:
QueryManagerRE::QueryS2_approach03
Returns:
vector<string>
Parameters:
(string tableName,string columnName,int streamSize);

Procedure:

1. Rowsvalues = findColValueForRows(All reows , columnName)

SELECT * FROM tableName WHERE columnName = elementName;
Returns:
vector<vector<string»
Name:
QueryManagerRE::QueryS_approach03
Parameters:
(string tableName,string columnName,string elementName,int streamSize);

Procedure:

1. Elementfound = findPosStringInDic(elementName, columnName)
2. Rowsfound = findPosValueInIndex(Elementfound)
3. Rowsvalues = findColValueForRows(Rowsfound , All columns)

4.5 4th approach: (No Dictionary) Benchmark approach

This last approach was done in order to have another Non-Buffering approach to compare with the previous one. Then, we can find out in which situations the B-tree Dictionary is powerful. This approach is quite similar to the first one, the Dictionary compression is undone and the elements are stored on disk directly with their real values and no compression. Non-Buffering from Windows is enabled while doing the queries.

4.5.1 DisklayerManager

The DiskLayerManager is quite similar to th one used in approach 01, described in section 4.2.1. The dictionary compression is undo and the real values of each column are saved.

The table is stored in two files ".data" and ".header". The header file contains information about each column, the name of the column and the length of the longest element; this is written in the first line. In the second line the number of rows of the table is written.

The data file contains the values of each column arranged in columnstore format. Each value is stored in such a way that it occupies in memory as much bytes as the longest. If one element is smaller padding is added.

4.5.2 QueryManager

In this approach 5 queries have been implemented, the same ones implemented in approach 03:

- SELECT * FROM tableName;
- SELECT columnName FROM tableName;
- SELECT * FROM tableName WHERE columnName = elementName;
- SELECT * FROM tableName WHERE columnName = elementName AND columnName2 = elementName2;
- SELECT columnName FROM tableName WHERE columnName = elementName;

In order to perform these queries 5 auxiliaries methods have been implemented. There are 2 methods that have been reused from approach 03. The following ones:

- openFileNoBuffering: this method open a file and avoids the buffering from windows when reading it. It returns the HANDLE to manage the file. Exactly the same used in approach 03.
- getColumnsInfo: this method returns and struct with information about the columns: the names, the length of the longest element, the position where each column begin in the dictionary file and the number of rows that the table has. This method is slightly different to the one used in approach 03 as different information is needed.

The other 3 auxiliary methods are related with the ones of approach 03 but the procedure is different:

- findPosValueInIndex: Find the relative positions where a value appears in the index. The search is performed in .data file. In this approach there is no dictionary compression due to this the search is done through the real values that could be quite long, consequently the search is slower than in approach 03. A vector with the positions of the rows found will be returned.
- findColValueForRows: This method retrieves the value of several rows for a specific column. The procedure is similar to the one in approach 3 but half of the process is skipped.In approach 03 this method requires two steps. First, it was needed to get the values of the rows in the index file, and then translate them with the dictionary file to get the real values. In this approach is only needed to get the values of the rows in .data file. In order to do this another auxiliary method has been implemented getThePositions.

- getThePositions: this method receives a vector of the relative positions of the rows in the column and it calculates their real position in the data file.

The method that has not been reused is findPosStringInDic, because there is not dictionary any more in this approach and no search is performed in dictionary file.

Queries implemented

In this approach 5 queries have been implemented, they are based in the auxiliary method explained above. The queries methods are a combination of the 2 main auxiliary methods:

- findPosValueInIndex
- findColValueForRows

SELECT * FROM tableName WHERE columnName = elementName AND columnName2 = elementName2;
Name:
QueryManagerRE2::QueryS5_approach04
Returns:
vector<vector<string»
Parameters:
(string tableName,string columnName,string elementName,string columnName2,string elementName2,int streamSize, string colSeleted,int colSel);

Procedure:

1. Rowsfound1 = findPosValueInIndex(elementName , columnName)
2. Rowsfound2 = findPosValueInIndex(elementName2 , columnName2)
3. Rowsvalues = findColValueForRows(Rowsfound1 U Rowsfound2 , All columns)

SELECT * FROM tableName;
Name:
QueryManagerRE2::QueryS4_approach04
Returns:
vector<vector<string»
Parameters:
(string tableName,int streamSize);

Procedure:

1. Rowsvalues = findColValueForRows(All rows , All columns)

SELECT columnSelected FROM tableName WHERE columnName = elementName;
Name:
QueryManagerRE2::QueryS3_approach04
Returns:

vector<string>
Parameters:
(string tableName,string columnName,string elementName,int streamSize,string columSelected);

Procedure:

1. Rowsfound = findPosValueInIndex(elementName , columnName)
2. Rowsvalues = findColValueForRows(Rowsfound , columnSelected)

SELECT columnName FROM tableName;
Name:
QueryManagerRE2::QueryS2_approach04
Returns:
vector<string>
Parameters:
(string tableName,string columnName,int streamSize);

Procedure:

1. Rowsvalues = findColValueForRows(All rows , columnName)

SELECT * FROM tableName WHERE columnName = elementName;
Returns:
vector<vector<string»
Name:
QueryManagerRE2::QueryS_approach04
Parameters:
(string tableName,string columnName,string elementName,int streamSize);

Procedure:

1. Rowsfound = findPosValueInIndex(elementName , columnName)
2. Rowsvalues = findColValueForRows(Rowsfound , All columns)

Chapter 5

Evaluation

In these chapter different sets of experiments has been performed in order to test the functionality and the behavior of every approach. The most relevant results obtained are presented in graphic in order to analyze them in a deepely. The motivation of each experiment is also discussed.

5.1 1st Approach

The main goal of this approach was fix the foundations of the project implement all classes and make them work together. Once we have achieved this we can go further and analyze in a deeper way how the IO operations affects the performance and justify the importance of go beyond and reduce the amount of these operations in order to reduce the time needed to process the queries. The table used in the two experiments is testA, the characteristics of this table is shown in TABLE 5.1.

The query executed in the two experiments is:

- SELECT * FROM testA WHERE col1= ”anaX”;

There are 2 coincidences in testA table.

5.1.1 First experiment: Different sizes of buffer - Query performs computation tasks and IO operations

In the first experiment the query has been tested with different sizes of buffer. In TABLE 5.2 the results are shown. The table contains the size of the buffer specified in bytes and the effective buffer used; it could be smaller than the buffer specified as the buffer has to retrieve full elements, therefore the effective

TABLE: testA	
Number of columns	*1*
Number of rows	*1,5 Millions*
Size	*8,58 MB (9.000.000 bytes)*
Longest element	*6 bytes*
Coincidences:	*2 coincidences*

Figure 5.1: Example table characteristics

Specified Buffer Size	Effective Buffer Size	Performance time - average of 10 repetitions
bytes	bytes	quantity
20	18	3,1841
30	30	3,177
40	36	3,094
50	48	3,0804
60	60	3,0808
70	66	3,0542
80	78	3,0339
90	90	3,047
100	96	3,0365
500	496	2,9933
1.000	996	2,9902
5.000	4.996	2,972
10.000	9.996	2,981
20.000	19.996	2,967
50.000	49.996	2,9712
100.000	99.996	2,9651
500.000	499.996	2,9534
1.000.000	999.996	2,9479
5.000.000	4.999.996	2,928

Figure 5.2: First experiment results

buffer size has to be multiple of the elements length. Finally the last column in the table is the time needed to perform the query and obtain the result, 10 repetitions of the queries has been done and the average is presented in seconds. Despite only 10 repetitions have been done is accurate enough for this experiment.

A graphic has been created with the results obtained in the experiment, the time needed to perform the query is displayed in Y axis and the effective buffer size in X axis; the graphic is presented in TABLE 5.3.

We can conclude that when the buffer is bigger and less IO operations are done less time is needed, despite the final amount of bytes read is the same for all the different buffer size, because, as there are not any optimizations the full file has to be read in order to find the coincidences.

We can observe that the difference of time needed between the smallest buffer tested and the biggest it is only 256,1 msec. We expected a bigger difference, as the smallest size is only 18 bytes and the biggest around 5 MB, this is due to the buffering done by the operating system. The first time we perform an IO operation against a file, the whole file is retrieve to the cache, as the operation system do know that is possible that more IO operations will be perform in this file, as it actually happens. But the results obtained won't be accurate and the optimizations implemented could not be properly tested if we allow the buffering from the operating system. Consequently in approach 3 and 4 it has been disabled.

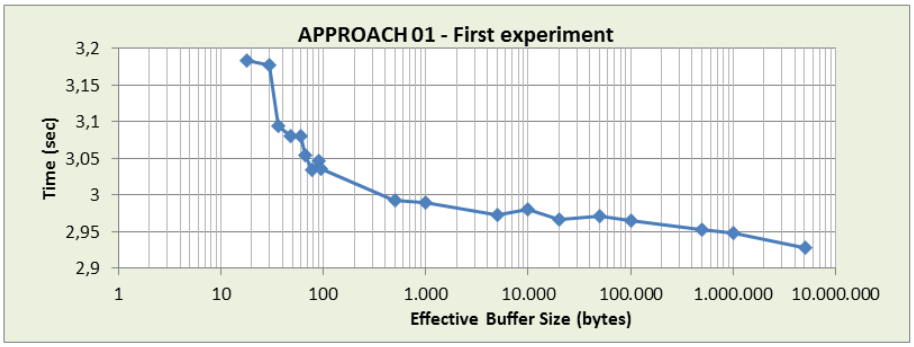


Figure 5.3: First experiment - Time VS Effective Buffer Size

5.1.2 Second experiment: Different sizes of buffer - Query performs only IO operations

As we have detected in the first experiment the IO operations are not significant in the time needed to perform the queries, and the most amount of time is needed to perform the computations tasks. The most important computation task in this query is compare every string of the column with the one specified by the request, "AnaX".

In this approach in order to analyze better the impact of the IO operations in the query the computations task has been skipped in this experiment, and the code related with computation has been disabled. It is important to be said that not only the string comparison is disabled, but also the retrieve of the values of other columns for the rows where "AnaX" is found. Despite only one column is available in this example, once we have found the row where "AnaX" is placed, we store these positions in a vector and then we read again the value of that column for that position, despite we already know is "AnaX". This is done in order to obtain more information about time in the tests. Anyway, in the second experiment this is disabled because not any coincidence will be found, but in the first experiment it was working.

In TABLE 5.4 the results are shown. The table contains the size of the effective buffer in bytes, the time needed to perform the query (only the IO operations are done), 10 repetitions has been done and the average is presented in seconds, and finally the number of IO operations done.

Two graphics has been created with the results obtained in the experiment, the first one shows the time needed to perform the query (Y axis) and the effective buffer size (X axis); this graphic is presented in figure 5.5. The second graphic shows the time needed to perform the query (Y axis) and the Number of IO operations needed (X axis); this graphic is presented in figure 5.6.

Now we compare the graphics obtained in the first and in the second experiment. As it can be observed in the second experiment the graphic is smoother, figure 5.5, this is due to the computation tasks are disabled, and as they needed around 3 seconds to perform their operations they could be affected by the load of the system. We can also check that the difference between the higher and the smaller buffer size is 102,4 msec that is less than the difference obtained in experiment one. This is due to the buffer size also affects the way compu-

Effective Buffer Size	Performance time - average of 10 repetitions	IO operations
bytes	seconds	quantity
18	0,1111	500.001
30	0,0697	300.001
36	0,0606	250.001
48	0,0477	187.501
60	0,0399	150.001
66	0,03711	136.364
78	0,0334	115.385
90	0,0297	100.001
96	0,028	93.751
496	0,0125	18.073
996	0,0108	9.037
4.996	0,0096	1.801
9.996	0,0094	901
19.996	0,0091	451
49.996	0,0087	181
99.996	0,0096	91
599.996	0,0087	19
999.996	0,0084	10
4.999.996	0,0087	2

Figure 5.4: Second experiment results

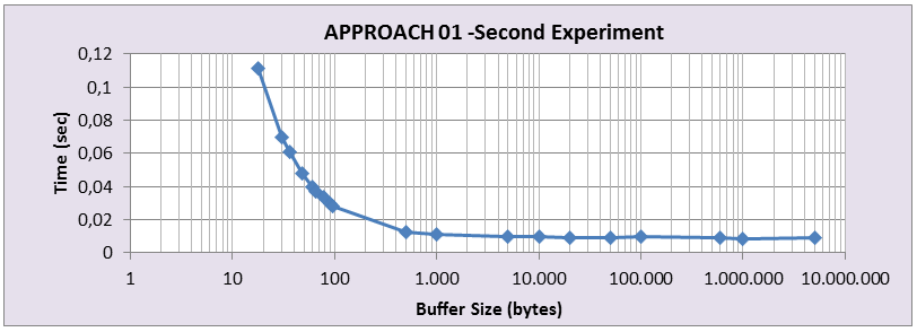


Figure 5.5: Second experiment - Time VS Effective Buffer Size

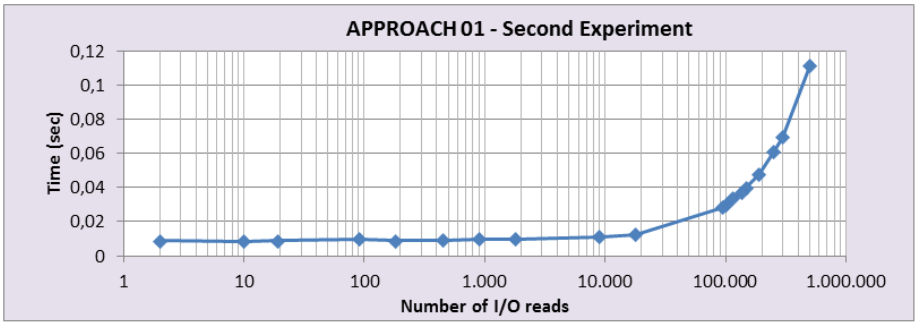


Figure 5.6: Second experiment - Time VS IO operations

TABLE: test1	
Number of columns	*3*
Number of rows	*100.000*
Size	*2,00 MB (2.099.999 bytes)*
Longest element	*6 bytes*

Figure 5.7: Example table test1 characteristics

tations tasks are performed, mainly the part of retrieving the other columns. Despite there is only one column its content is retrieved for the rows found, as it has been explained, but in this IO operation the buffer size available is used despite the amount of data needed is small; and one IO operation is done for each position found. In the second experiment this functionality is disabled.

We can conclude from graphic presented in figure 5.6 that the time needed is constant until certain amount of IO operations but there is a threshold where the time needed start to increase. Thanks to this first experiments we can determine that one of the main goals of this project is reduce the number of IO operations to keep it lower than the threshold value.

5.2 2nd Approach

The main goal of first approach was setting the baseline of the project. Once we have achieved this we can go further and analyze in a deeper way how the IO operations affects the performance and justify the importance of go beyond and reduce the amount of these operations in order to reduce the time needed to process the queries.

5.2.1 First experiment

In this experiment table test1 has been used. The characteristics of this table are shown in figure 5.7.

The main objective of this experiment was testing the functionality of the different queries implemented. The three queries executed were:

- Q1 - SELECT * FROM test1 WHERE col1 = "100001"; -> 7 matches
- Q2 - SELECT col1 FROM test1;

Specified Buffer Size	Q1	Q2	Q3
bytes	seconds	seconds	seconds
1000	0.0002	15.498	47.022
10000	0.0002	13.711	42.381
100000	0.0002	13.395	38.797

Figure 5.8: First experiment results

TABLE: testA	
Number of columns	*1*
Number of rows	*1,5 Millions*
Size	*8,58 MB (9.000.000 bytes)*
Longest element	*6 bytes*

Figure 5.9: Example table testA characteristics

- Q3 - SELECT * FROM test1;

The experiment has been performed for different buffer sizes, but as buffering from Windows was enabled the results do not give useful information. The results obtained are shown in table 5.8. The most important conclusions that can be obtained are that the functionalities were properly achieved and that the buffering from Windows should be disabled to be able to test properly the program.

5.2.2 Second experiment: Different sizes of buffer - Query performs computation tasks and IO operations

The table used in this experiment is the same one used in the first experiment of approach 01, section 5.1.1. The table is testA, the characteristics of this table are shown in figure 5.9.

The main objective of this experiment is checking the impact of the B-tree 1 level in the total time needed to perform the query. The query executed in the experiment is: SELECT * FROM testA WHERE col1= ”anaX”; There are 2 matches. The query has been tested for different buffer sizes the results obtained are presented in table 5.10 and in Figure 5.11.

In the graphic displayed in figure 5.11 it can be seen that the results are not very accurate. This is due to the execution time has been highly reduced thanks to the B-tree 1-level and then it has more importance the load of the system and the buffering perform by Windows due to this the graphic obtained is abrupt.

5.3 3rd Approach

The main goal of testing this approach is to analyze the performance achieved. We are going to test different queries in order to determine in which ones this approach gets the best results. Two versions of this approach have been implemented as it has been detailed in section 4.4; we are going to do the same tests in both versions to determine which version fits better in each scenario. The

Specified Buffer Size	Effective Buffer Size	Performance time - average of 10 repetitions
bytes	bytes	quantity
20	18	0,0561
30	30	0,0156
40	36	0,0109
50	48	0,0125
60	60	0,0094
70	66	0,0078
80	78	0,0078
90	90	0,0078
100	96	0,0094
500	496	0,0062
1.000	996	0,0078
5.000	4.996	0,0063
10.000	9.996	0,0062
20.000	19.996	0,0047
50.000	49.996	0,0063
100.000	99.996	0,0078
500.000	499.996	0,0047
1.000.000	999.996	0,0047
5.000.000	4.999.996	0,0078

Figure 5.10: Second experiment results - table

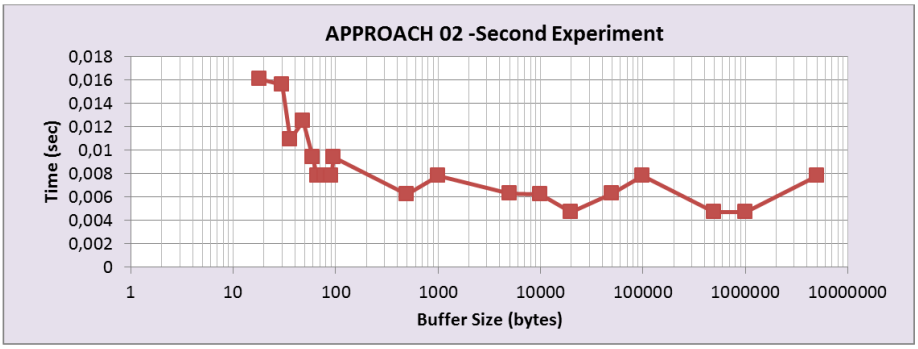


Figure 5.11: Second experiment results - graphic

TABLE: lineitem		
Number of columns		*16*
Number of rows		*6.001.215*
Size		*740 MB*
Order	Column name	Dictionary elements
1	L_ORDERKEY	1.500.000
2	L_PARTKEY	200.000
3	L_SUPPKEY	10.000
4	L_LINENUMBER	7
5	L_QUANTITY	50
6	L_EXTENDEDPRICE	933.900
7	L_DISCOUNT	11
8	L_TAX	9
9	L_RETURNFLAG	3
10	L_LINESTATUS	2
11	L_SHIPDATE	2.526
12	L_COMMITDATE	2.466
13	L_RECEIPTDATE	2.554
14	L_SHIPINSTRUCT	4
15	L_SHIPMODE	7
16	L_COMMENT	4.580.667

Figure 5.12: Table lineitem characteristics

table used in all the experiments is lineitem, the characteristics of this table and the properties of their columns are described in 5.12. It is important to take in to account that in this approach the buffering from windows is disabled, so it is not possible to compare with the results obtained in previous approaches, as the times obtained will be obviously higher.

5.3.1 First experiment: No conditional version

The version without the conditional case has been first tested. In this experiment three queries have been checked:

1. SELECT * WHERE L_COMMITDATE = ”1993-08-16”
2. SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
3. SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ”, quick deposits. ironic, unusual deposi;”

These queries have the following number of coincidences:

1. 2454 elements found
2. 15 elements found
3. 1 element found

Buffer Size	Query 1	Query 2	Query 3	Total (100 rep. x 3Q)
bytes	sec	sec	sec	sec
4096	34,512	6,131	2,082	4272,61
8192	35,025	5,492	1,385	4190,26
16384	33,725	5,300	1,031	4005,65
32768	31,884	4,828	0,742	3745,37
65536	28,421	4,842	0,612	3387,65
131072	25,788	5,011	0,467	3126,65
262144	23,414	6,385	0,740	3053,92
524288	21,533	8,131	0,416	3008,03
1048576	19,743	8,997	0,755	2949,53

Figure 5.13: APPROACH03: First experiment results

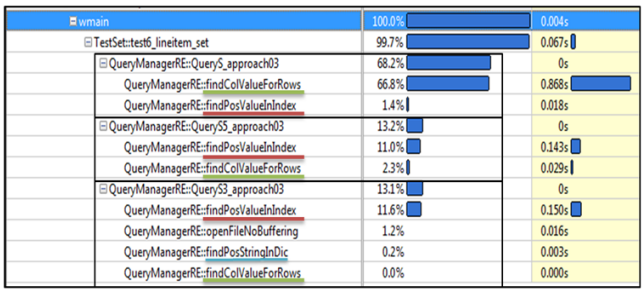


Figure 5.14: VTune test results

The queries have been tested for different buffer sizes, the experiment begin with a buffer of 4096 bytes and it is increased by a factor of 2 until around 1MB. Each query has been executed 100 times and the average has been calculated. The results obtained are presented in figure 5.13, the size of the buffer is specified, the time needed for each query and the total time needed to perform the full experiment, 100 repetitions for each query. Also the results are presented in figure 5.13.

It can be observed that the behavior is different for each of the queries. It is going to be detailed the reason for these different results. The main point is that the auxiliary method that consumes more execution time is different in each query. The key auxiliary methods are described in section 4.4.2, they are the following ones:

- findPosStringInDic
- findPosValueInIndex
- findColValueForRows

In order to understand well the results obtained the tool VTune from Intel have been used and we have obtained the results presented in figure 5.14. Then, it can be determined which is the auxiliary method that requires more time. This tool allows us to explain accurately the differences of the 3 queries.

APPROACH 03: No conditional, Query 1

- SELECT * WHERE L_COMMITDATE = ”1993-08-16”

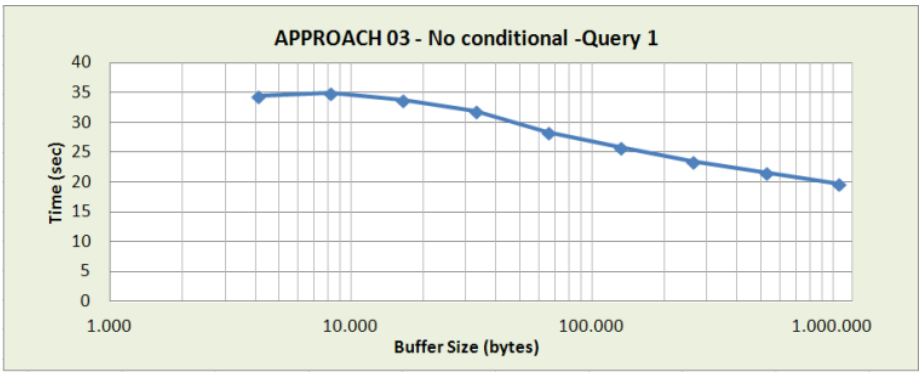


Figure 5.15: APPROACH 03: No conditional, Query 1

- -> 2454 elements found

In this query the auxiliary method that consumes more execution time is findColValueForRows, as there 2454 coincidences many IO operations are needed in order to retrieve the attributes of the rows found, 16 columns in total. We have to take in to account that elements are sorted and when they are close enough in the dictionary (that several values to translate are available in one buffer read), the translation of their values (from the index to the dictionary) requires less IO operations as it was explained in section 4.4.2. Therefore the impact of increasing the buffer size is quite significant. As we can check in table 5.13 and in figure 5.15 each time the buffer size is double the execution time is reduce 2 seconds in average.

The next auxiliary method that requires more execution is findPosValueInIndex and then findPosStringinDic but they are not significant.

APPROACH 03: No conditional, Query 2

- SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
- -> 15 elements found

In this query the auxiliary method that consumes more execution depends on the size of the buffer of the buffer.

- When the size of the buffer is small the auxiliary method that consumes more execution time is findPosValueInIndex. The table has 6 millions of rows; each element in the index occupies 4 bytes and the number of IO operations that requires this method for a buffer size of 4096 bytes is 5860 (6 millions x 4bytes / 4096 bytes). Method findColValueForRows requires less execution time as there are only 15 coincidences and in the worst case the maximum number of IO operations will be 450 (15 coincidences x 15 columns x 2 for translate) . If we take in to account that there are 8 columns that have a small dictionary then they will always fit in one buffer read, consequently the number of IO operations needed in the worst case will be 338 (15 coincidences x 8 columns x 2 for translate + 15 coincidences x 7 columns + 7 for translate)

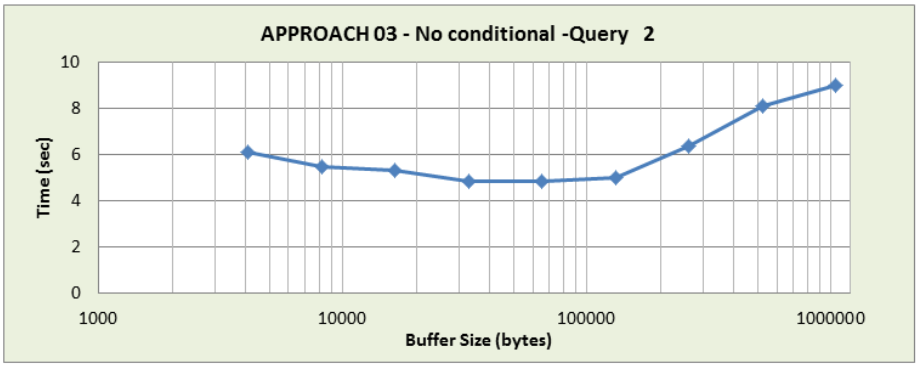


Figure 5.16: APPROACH 03: No conditional, Query 2

- When the size of the buffer is big the auxiliary method that consumes more execution time is findColValueForRows. This method will need around 300 IO operations it could be more or less in dependence of the number of dictionaries that requires only one buffer to be read or if the coincidences are close enough to require only one IO operation to translate the 15 values. Method findPosValueInIndex will require less IO operations when the buffer size is increased, the number of IO operation will decrease in a linear way. For example, the method will require only 23 IO operations for a buffer size of 1048576 bytes.

The problem that arises in this query is the fact that number of IO operations required by findColValueForRows method could keep constant while the buffer size is increased, because the number of coincidences is small and the sorting optimization could has no effect. Consequently, the number of bytes read will be dramatically increased and despite the number of IO operations required by findPosValueInIndex method is reduced a lot, from 5.860 to 23, the total query execution time will be increase because of the huge amount of bytes retrieved by findColValueForRows method, as it actually happens in this experiment and it could be checked in figure 5.16.

An estimation of the IO operations needed and the amount of bytes read by each method is presented in figure 5.17. (The values of findPosValueInIndex are accurate but the ones of findColValueForRows has been estimated, probably it could not happen in this scenario that 314MB have been read by findColValueForRows method, as the table shows, it could have read to up to 100MB, but in an scenario were the columns are similar to L_COMMENT one, where the dictionary compression is low and the element have a big length, this table could be realistic) A solution to this situation has been described in futures work chapter.

APPROACH 03: No conditional, Query 3

- SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"

- -> 1 elements found

Buffer Size	findPosValueInIndex (Real)		findColValueForRows (Estimation)	
bytes	bytes read	IO operations	bytes read	IO operations
4096	24.000.000	5.860	1.384.848	338
8192	24.000.000	2.930	2.769.296	338
16384	24.000.000	1.465	5.538.192	338
32768	24.000.000	733	11.075.984	338
65536	24.000.000	367	19.661.200	300
131072	24.000.000	184	39.322.000	300
262144	24.000.000	92	78.643.600	300
524288	24.000.000	46	157.286.800	300
1048576	24.000.000	23	314.573.200	300

Figure 5.17: APPROACH 03: Auxiliary methods: Bytes read and IO operations

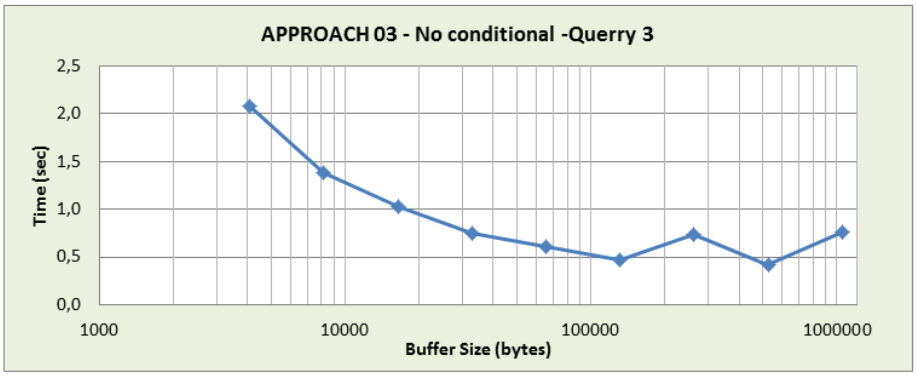


Figure 5.18: APPROACH 03: No conditional, Query 3

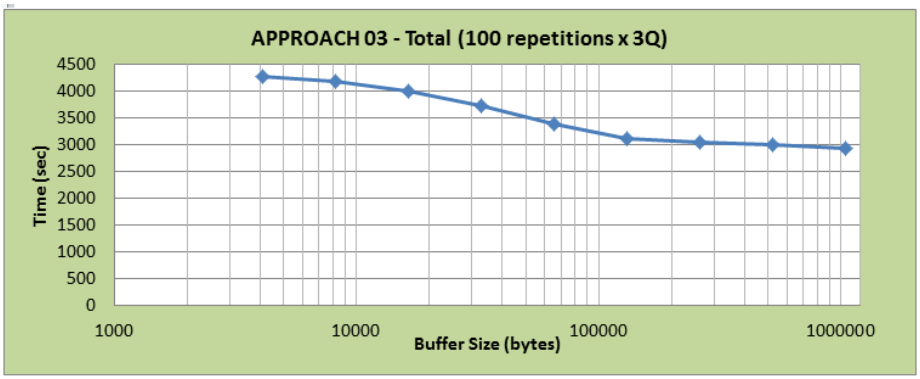


Figure 5.19: APPROACH 03: No conditional, Complete Experiment

This query is a good experiment to test the powerfully of the B-tree implementation. In this scenario one value is searched in a dictionary of 4.580.667 elements and they are long character strings. Despite the dictionary of this column is quite big, the amount of IO operations required to perform the search is always small. When the buffer size is 4096 the amount of IO operations needed will be only 4, and only 16.384 bytes will be read (4 x 4096). Therefore, the auxiliary method that consumes more execution time is findPosValueInIndex. When the buffer size is increased the IO operations needed by findPosValueInIndex method decrease in a linear way (but the amount of bytes read in each operation is increased).

Thanks to this behavior the execution time of this auxiliary method is decrease, and the total execution time of the query is decreased. But the IO operations required by findPosStringInDic are slightly reduced when the buffer size is increased, 4 with 4096 and to 2 with 1MB, but the amount of bytes read are highly increased, from 16.384 to 2MB. Then we can conclude that when the buffer size is big enough when we increase it more the time required by method findPosValueInIndex remains more or less constant, but the time time required by method findPosStringInDic is increased. Consequently the graphic of figure 5.18 shows how the total execution time of the query increases after a 0,5MB.

There is a unusual value in the graphic of figure 5.18 for a buffer size of 0,25MB, this is due to the element searched, in this situation, is a root element in the B-tree, and 1 IO operation is skipped of 0,25MB read.

APPROACH 03: Complete experiment

- 100 X SELECT * WHERE L_COMMITDATE = ”1993-08-16”
- 100 X S SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
- 100 X S SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ”, quick deposits. ironic, unusual deposi;”

Once we have analyzed the results of each query we have to analyze the times obtained for the global experiment, which means the execution of 300 queries against the files. In figure 5.19 a graphic of the results is presented. It can be

noticed that the times always decrease, this is due to the dominant query is the first one, and its execution time decreases with the increment of the buffer size. Due to the behavior of the other two queries the slope is smaller after a buffer size of 0,131MB.

The conclusions that we can obtain from this experiment is that a big buffer size could not be appropriate for all the situations. In this case, for findColValueForRows method, is not necessary a big buffer size when the number of coincidences is small as it has been explained in section 5.3.2. And for findPosStringInDic method, a big buffer size is not needed when our B-tree already have a small number of level. A solution is proposed related with this behavior in Future Work chapter, in section 7.1.

5.3.2 Second experiment: Conditional version

In this experiment the version with the conditional case has been tested. It has been evaluated the same three queries used in the first experiment:

1. SELECT * WHERE L_COMMITDATE = "1993-08-16"
2. SELECT * WHERE L_COMMITDATE = "1993-08-16" AND L_RECEIPTDATE = "1993-10-28"
3. SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"

These queries have the following number of coincidences:

1. 2454 elements found
2. 15 elements found
3. 1 element found

In this experiment the queries have also been tested for different buffer sizes, the experiment begin with a buffer of 4096 bytes and it is increased by a factor of 2 until around 1MB. Each query has been executed 100 times and the average has been calculated. The results obtained are presented in figure 5.20, the size of the buffer is specified, the time needed for each query and the total time needed to perform the full experiment, 100 repetitions for each query.

In this approach the conditional case is enabled, as it has been explained in section 5.3.2 the method procedure changes when the full dictionary fits in the buffer. In order to analyze properly the results it is important to study the number of dictionaries that fits in the buffer for each size. This study has been done and the results are presented in Table 5.21. The table also presents the deepness of the tree; this means the number of levels behind the roots. This number plus 1 specifies the number of IO operations required to perform a search in the dictionary in the worst case. Also the total number of columns which dictionary fits in a buffer is presented in the bottom of the table. A graphic has been created with the total values, and it is presented in figure 5.22.

Another graphic that describes the percentage of columns which dictionary fits in a buffer is presented in figure 5.23. The percentage is displayed for each buffer size value. The main conclusion that we can obtain from this study is

Buffer Size	Query 1	Query 2	Query 3	Total (100 rep. x 3Q)
bytes	sec	sec	sec	sec
4096	28,760	7,799	2,091	3864,99
8192	30,398	6,848	1,263	3850,77
16384	28,844	6,247	0,941	3560,07
32768	26,844	4,857	0,707	3240,78
65536	24,419	5,870	0,555	3084,44
131072	22,746	5,680	0,463	2888,94
262144	20,724	6,328	0,371	2742,19
524288	18,471	6,262	0,322	2505,47
1048576	16,994	7,913	0,599	2550,63

Figure 5.20: APPROACH03: Second experiment results

Dictionaries		Buffer Size															
		4096		8192		16384		32768		65536		131072		262144		524288		1048576	
Column Name	Size	D	F	D	F	D	F	D	F	D	F	D	F	D	F	D	F	D	F
L_ORDERKEY	1.500.000	2	0	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
L_PARTKEY	200.000	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
L_SUPPKEY	10.000	1	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1
L_LINENUMBER	7	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_QUANTITY	50	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_EXTENDEDPRICE	933.900	2	0	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
L_DISCOUNT	11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_TAX	9	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_RETURNFLAG	3	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_LINESTATUS	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_SHIPDATE	2.526	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1
L_COMMITDATE	2.466	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1
L_RECEIPTDATE	2.554	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1
L_SHIPINSTRUCT	4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_SHIPMODE	7	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
L_COMMENT	4.580.667	3	0	2	0	2	0	2	0	2	0	1	0	1	0	1	0	1	0
	Total	13	8	11	8	9	8	9	8	6	11	4	12	4	12	4	12	4	12

Figure 5.21: APPROACH03: First experiment results

that the percentage keeps constant from 4096 to 32768 and from 0,13MB to 1MB. But is important that we take in to account that varies from 32768 to 0,13MB and this affects the experiment, this values are boxed in figure 5.22.

APPROACH 03: Conditional, Query 1

- SELECT * WHERE L_COMMITDATE = ”1993-08-16”
- -> 2454 elements found

The graphic obtained for this query is similar to the one obtained in the first experiment, it is presented in figure 5.24. findColValueForRows method is the one that consumes more execution time also, as it has been explained in section 5.3.2. The buffer size affects the time in a similar way and each time the buffer size is double the execution time is reduce 2 seconds in average.Auxiliary methods findPosValueInIndex and findPosStringinDic have a not significant execution time.

The main conclusion obtained from query 1 is that the time is reduced 4 seconds in average if we compare it with the non-conditional case, this is because

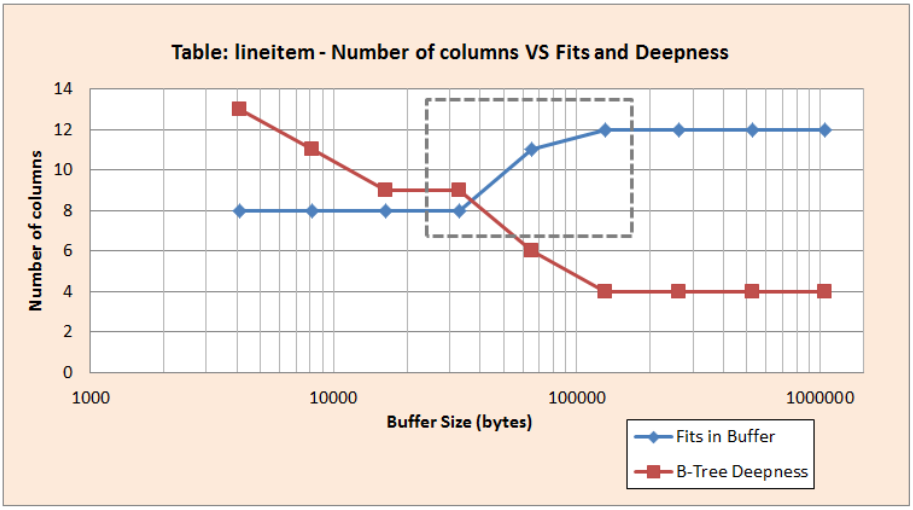


Figure 5.22: Table linitem: Dictionaries deepness and fits in one buffer

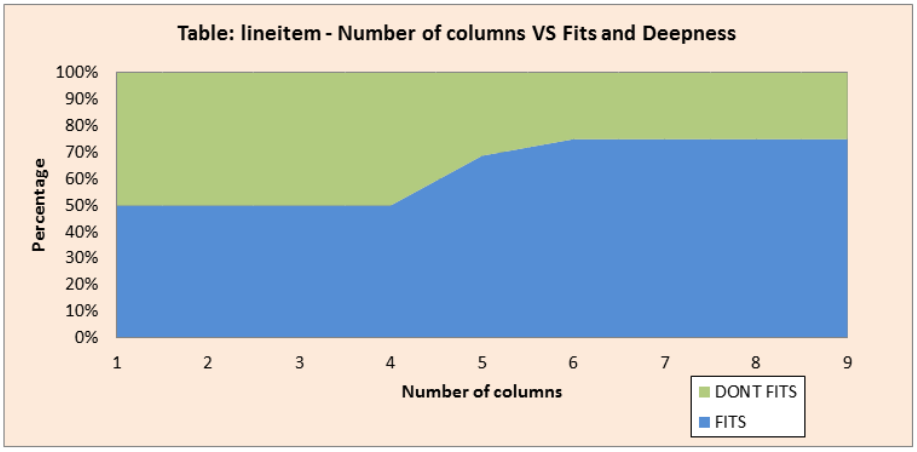


Figure 5.23: Table linitem: Percentage of dictionaries that fits in one buffer

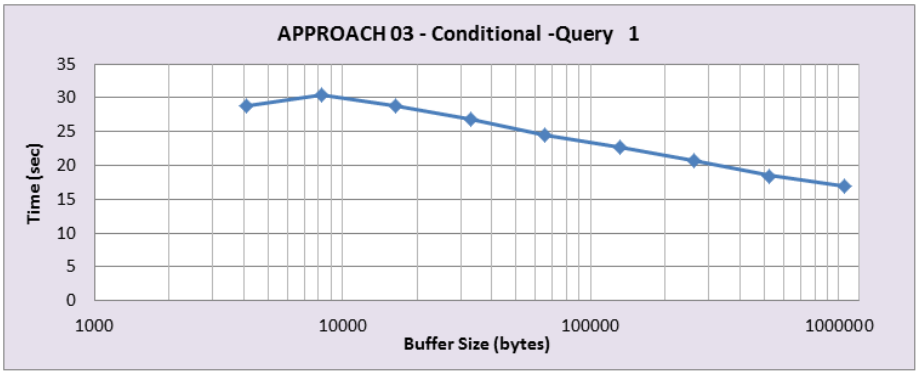


Figure 5.24: APPROACH 03: Conditional, Query 1

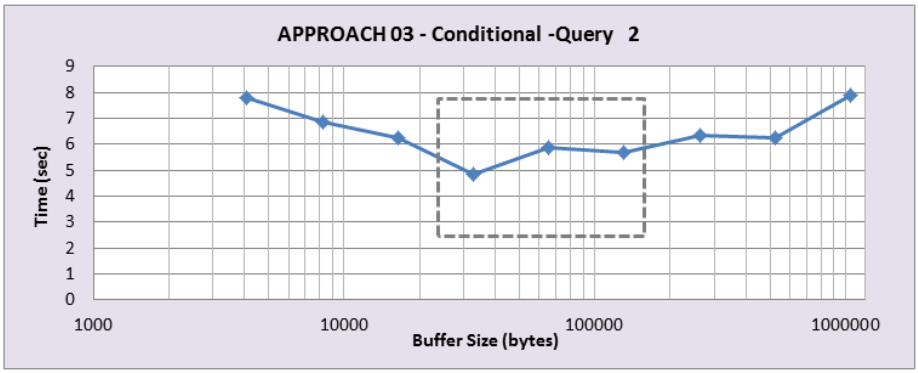


Figure 5.25: APPROACH 03: Conditional, Query 2

there is a big amount of coincidences and even for the smallest buffer used there are 8 dictionaries that fits in it and there is no needed of sorting for 8 vectors of 2454 elements For the biggest buffer there are 12, only 3 more, due to this the impact is not so big, and there are not abrupt values when the number

APPROACH 03: Conditional, Query 2

- SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
- -> 15 elements found

The behavior obtained for this query could be explained in the same way as it has been done in section 5.3.2. But there are abrupt changes due to the percentage of dictionary that fits in one buffer as it has been introduced in the beginning of the experiment. These values are boxed in figure 5.25, they are for buffer sizes from 32768 to 0,13MB. The percentage changes from 50% to 75%.

APPROACH 03: Conditional, Query 3

- SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ”, quick deposits. ironic, unusual deposi;”
- -> 1 elements found

In this query the conditional case has no effect because there is only one element found and there is no need of sorting. Consequently the behavior is the same explained in the first experiment, section 5.3.2. The results obtained are displayed in figure 5.26.

APPROACH 03: Complete experiment

- 100 X SELECT * WHERE L_COMMITDATE = ”1993-08-16”
- 100 X S SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”

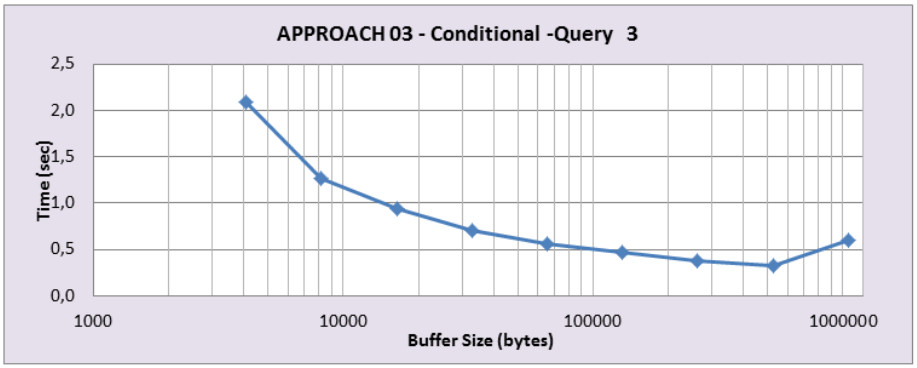


Figure 5.26: APPROACH 03: Conditional, Query 3

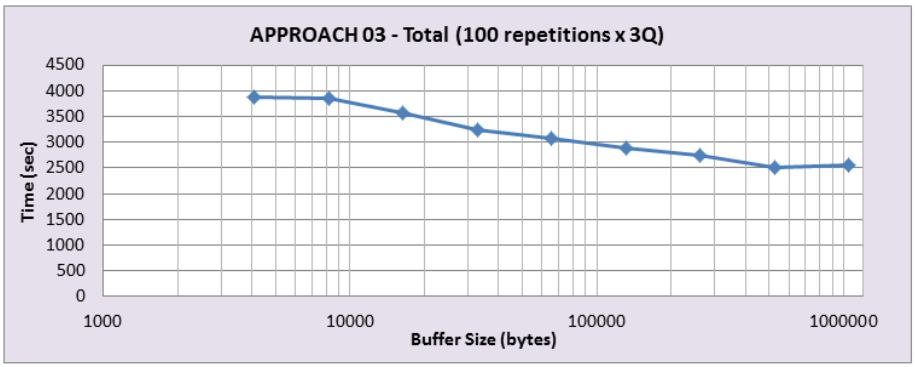


Figure 5.27: APPROACH 03: Conditional, Complete Experiment

- 100 X S SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"

The result of the complete experiments are displayed in figure 5.27. The main conclusions are the same obtained in the first experiment. And a new conclusion has to be added, the behavior of the query is quite related with the number of coincidences and with the number of columns which dictionary that fits in a buffer. A table is presented in figure 5.28 to summarize how the conditional case affects the query execution time in dependence with percentage of dictionaries that fits and the number of coincidences.

Percentage Fits / Coincidences	0 % - 50 %	50% - 60 %	60% - 100%
Query 1 - 2454	Very Significant	Very Significant	Very Significant
Query 2 - 15	Few Significant	Significant	Significant
Query 3 - 1	Not affects	Not affects	Not affects

Figure 5.28: Impact of the conditional case in the execution time of the query

Buffer Size	Query 1	Query 2	Query 3	Total (100 rep. x 3Q)
bytes	sec	sec	sec	sec
4096	5,7523	-1,66745	-0,00849	407,62
8192	4,6274	-1,35563	0,12293	339,49
16384	4,8808	-0,94727	0,09032	445,58
32768	5,0394	-0,02898	0,03536	504,59
65536	4,0023	-1,02854	0,05649	303,21
131072	3,0421	-0,66894	0,004	237,71
262144	2,6901	0,05736	0,36988	311,73
524288	3,0626	1,86887	0,09429	502,56
1048576	2,7488	1,08326	0,15688	398,9

Figure 5.29: APPROACH 03: Non conditional version VS Conditional version - time differences

5.3.3 Non conditional version VS Conditional version

Once the results for the two versions have been obtained, both experiments must be compared to determine which version has better performance. In order to do this the values obtained has been subtracted and the results are presented in Table 5.29, the times showed are "No Conditional" version minus "Conditional". Also the graphics obtained for each experiment are presented together in figure 5.30. Now the results are analyzed for each query in order to choose the best version, and finally determine which the best solution for a random workload is.

APPROACH 03: No conditional VS Conditional, Query 1

- SELECT * WHERE L_COMMITDATE = "1993-08-16"
- -> 2454 elements found

In this query the biggest difference between the two versions is obtained. This is due to the sorting of the elements found is skipped when the dictionary of that column fits in the buffer. The sorting usually is a good optimization and helps to avoid many IO operations, but when the dictionary fits in the buffer only one IO operations will be done for sure and no sorting is needed. This is shown in the first graphic of figure5.30.

In this query the number of element found is 2454 consequently avoid the sorting of this elements when is not needed improves the performance as it has been explained in section XX and 4 seconds are reduced in average from No conditional version to Conditional one. It can be concluded that when the number of coincidences is high and there are many columns to be retrieve which dictionaries fits in one buffer the performance of the Conditional version is better.

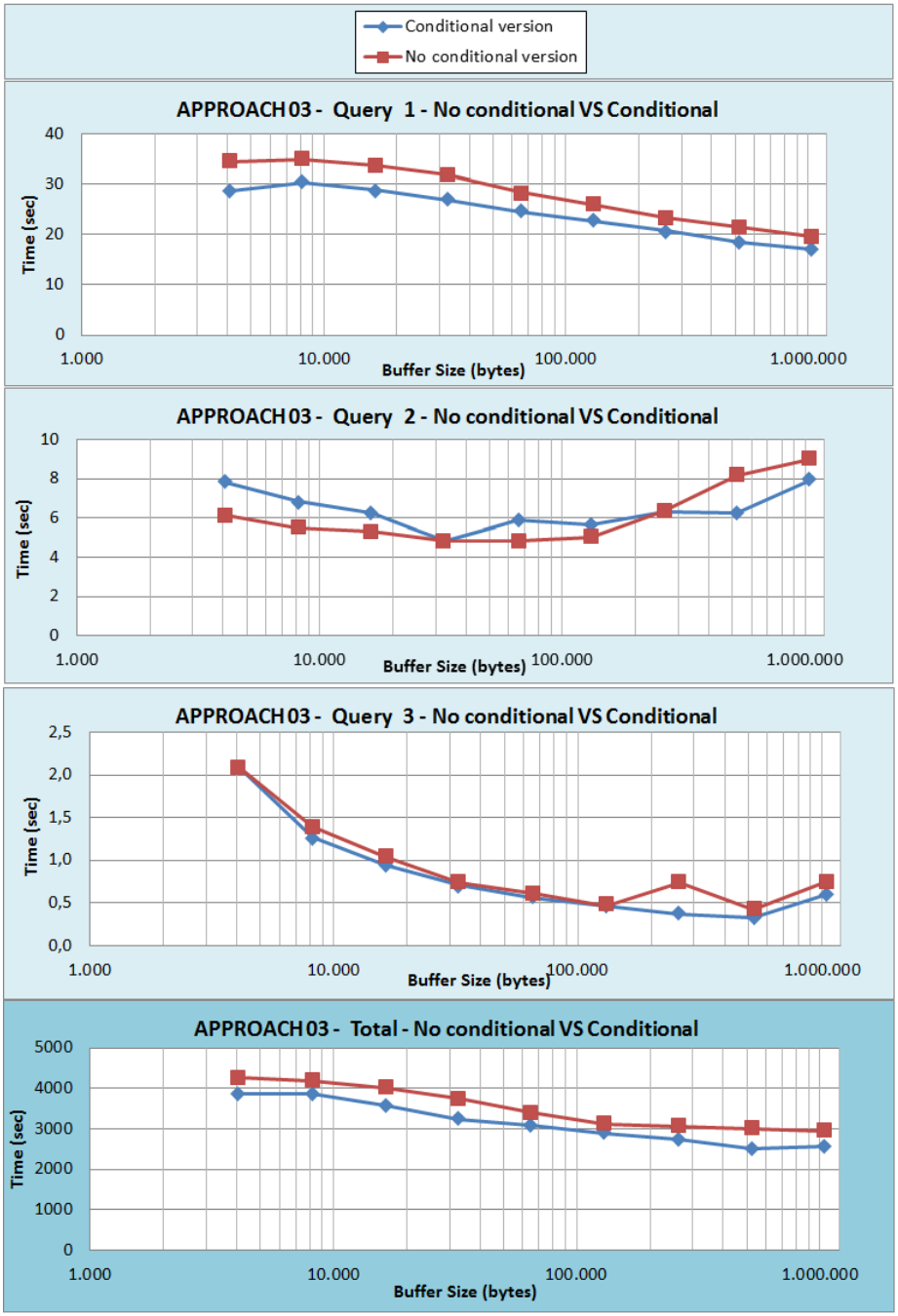


Figure 5.30: APPROACH 03: Non conditional version VS Conditional version - graphics

APPROACH 03: No conditional VS Conditional, Query 2

- SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
- -> 15 elements found

In this query the auxiliary method that consumes more execution time is related to the buffer size. The method that requires more time could be findPosValueInIndex or findColValueForRows. The method findPosValueInIndex requires the same time in both versions. But findColValueForRows is differrent for each version.

When the buffer size is small and the number of coincidences also (only 15 in this example) the computation time needed to retrieve the value from the array read is more significant than the sorting of only 15 elements, due to this when the sorting is done the retrieve of the value from the array could be skipped when there are two equal elements in the other columns of the rows where the coincidences are. But when the buffer is bigger as there are many dictionaries that fit there are many sortings skipped. This is shown in the second graphic of figure5.30.

It can be determined that No conditional version has better performance after a buffer size of 0.13 MB, that is a most common buffer size value than the lower ones.

APPROACH 03: No conditional VS Conditional, Query 3

- SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ”, quick deposits. ironic, unusual deposi;”
- -> 1 elements found

In this experiment the auxiliary method that requires more execution time is findPosStringInDic, and method findColValueForRows, the one where the conditional case is activated, is not used. Due to this the beahviour is the same, but for an outlier value. There is a slight difference due to the load of the system when performing the experiments. This is shown in the third graphic of figure5.30.

APPROACH 03: No conditional VS Conditional, Total

- 100 X SELECT * WHERE L_COMMITDATE = ”1993-08-16”
- 100 X S SELECT * WHERE L_COMMITDATE = ”1993-08-16” AND L_RECEIPTDATE = ”1993-10-28”
- 100 X S SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ”, quick deposits. ironic, unusual deposi;”

For the full experiment, a workload of 300 queries, the average difference is 384 sec. This is shown in the fourth graphic of figure5.30. This is due to the dominant query is Querry 1 and it has a bigger impact in the whole experiment than the other two queries together. This query has a much better performance in the Conditional version than in the No conditional one. Query 3 has the

same behavior in both version and for Query 2 Conditional version has better performance when the buffer is higher than 0,13 MB. The we can conclude that Conditional version is better for a random workload, and in some little specific scenarios No Conditional version is better, only when the buffer size is small, the queries demands the values of other columns for the coincidences found and there are not many elements founds.

5.4 4th Approach

The main goal of testing this approach is compare a noncomplex solution with the optimized solution. In approach 03 some optimizations has been implemented. The optimizations are: B-tree format for the Dictionary, sorting the elements to avoid IO readings, write in the index file the position of the value in the dictionary file for translating a conditional case to escape sorting when not needed and the last one is dictionary compression that the table has. But in approach 04, the dictionaries are undone and the table is stored with its real values. The B-tree optimization cannot be used, nor writing the directly the position in the index, because there are not index file nor dictionary file anymore. Only data file is available. The sorting and the conditional case have been used, because these two optimizations can be implemented in this solution and improve a lot the performance. If they are no used there is no possible benchmarking as this solution would be much slower.

5.4.1 First experiment: Approach 03 VS Approach 04

The table used for this experiment and the queries tested are the same that have been evaluated in approach 03, experiment 1st and 2nd. The table is "lineitem" and the queries are:

1. SELECT * WHERE L_COMMITDATE = "1993-08-16"
2. SELECT * WHERE L_COMMITDATE = "1993-08-16" AND L_RECEIPTDATE = "1993-10-28"
3. SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"

The results obtained are displayed in table 5.31 and in figure 5.32.

APPROACH 03 Conditional VS APPROACH 04, Query 1

- SELECT * WHERE L_COMMITDATE = "1993-08-16"
- -> 2454 elements found

In this query the auxiliary method that consumes more execution time and clearly dominant is findColValueForRows, as it happened in approach 03. The sorting optimization and the conditional case is activated in both approaches, but in approach 03 the performance is improved when the buffer is bigger, as it was explained in section 5.3.1, but for this approach not. This is because in approach 03 the sorting optimization is quite powerful thanks to the dictionary compression, and when the buffer is increased there are many IO operations

that are skipped. But when there is not dictionary compression is more difficult to avoid IO operation thanks to sorting, and it is quite hard for column "L_COMMENT" which elements are quite large.

It can be concluded that the performance of Approach 03 Conditional version is better, only for very small buffer sizes, when sorting is not effective, they present the same performance.

APPROACH 03 Conditional VS APPROACH 04, Query 2

- SELECT * WHERE L_COMMITDATE = "1993-08-16" AND L_RECEIPTDATE = "1993-10-28"
- -> 15 elements found

The results obtained in the experiment for this query are displayed in the second graphic of figure 5.32. The execution time is improved up to 2 seconds for a buffer size of 32768 bytes in Approach 03. But when the buffer is increased the results of the two approaches became close.

APPROACH 03 Conditional VS APPROACH 04, Query 3

- SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"
- -> 1 elements found

In this query each approach works in a different way. Approach 03 performs a search in the dictionary file and then performs a second search in the index file, but the second search is through ints values. Despite the dictionary has a very bad compression (4.580.667 elements) and the length of the elements is large, only 4 IO operations are required for the smallest buffer size an 2 for the biggest, thanks to the B-tree (without the B-tree first search would need a quite significant amount of time).

In approach 04 only one search is performed but through a file of large strings, due to this the performance is much worse in this approach than in approach 03. The results are shown in the third graphic of figure 5.32.

APPROACH 03 Conditional VS APPROACH 04, Total

- 100 X SELECT * WHERE L_COMMITDATE = "1993-08-16"
- 100 X S SELECT * WHERE L_COMMITDATE = "1993-08-16" AND L_RECEIPTDATE = "1993-10-28"
- 100 X S SELECT L_SHIPINSTRUCT WHERE L_COMMENT = ", quick deposits. ironic, unusual deposi;"

It has been described in all the queries studies that performance is better in approach 03. Due to this it can be determine that optimizations implemented are efficient and are able to reduce the global time of the experiment up to 35 minutes in average, for the different buffer sizes.

Buffer Size	Query 1	Query 2	Query 3	Total (100 rep. x 3Q)
bytes	sec	sec	sec	sec
4096,0000	30,6568	9,2593	22,5724	6248,8500
8192,0000	30,2047	7,9707	17,4376	5561,3000
16384,0000	31,7000	7,3839	15,4408	5452,4700
32768,0000	31,4928	7,7068	13,3074	5250,7000
65536,0000	31,4388	6,8372	12,2519	5052,8000
131072,0000	32,2094	6,5532	11,2616	5002,4200
262144,0000	31,7541	6,2981	10,6418	4869,4000
524288,0000	31,3825	6,6019	10,5737	4855,8100
1048576,0000	30,6928	8,1387	10,0836	4891,5100

Figure 5.31: APPROACH 03 VS APPROACH 04 - results

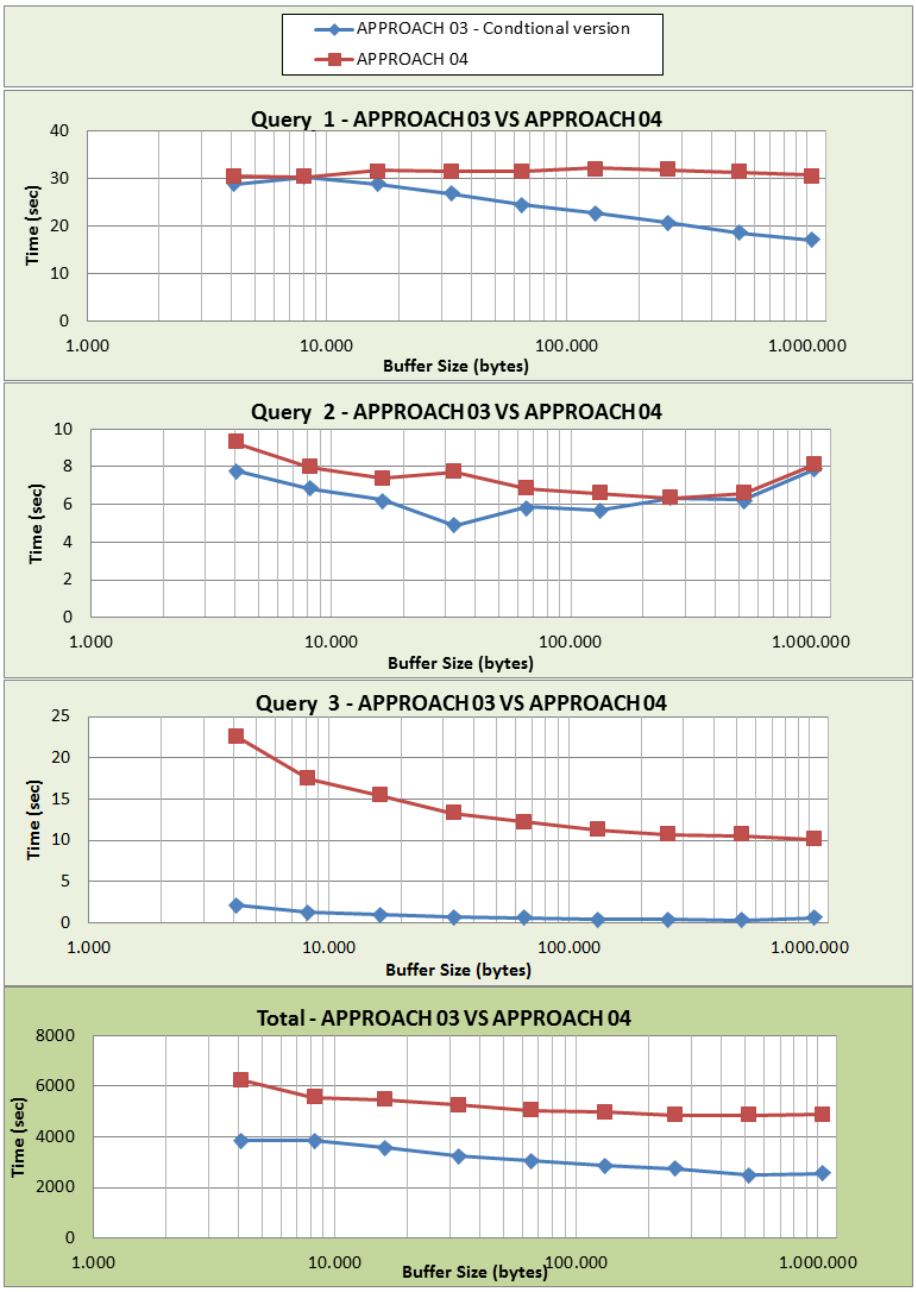


Figure 5.32: APPROACH 03 VS APPROACH 04 - graphics

Chapter 6

Conclusions

The main conclusions obtained in this thesis can be summarized in three sections. The first one is that the problem has been recognized and fragmented, the main functionality areas where the optimizations must be done has been identified. The second main conclusion is how the buffer size selected and the dictionary compression affects the different areas identified. And finally the last section explains which are the optimizations implemented in this thesis and their target areas.

6.1 Fragmentation of the problem

In approach 02 many lines of code were repeated in the four queries that class QueryManager provides. Due to this, auxiliary methods where created when implementing the queries desired for approach 03. Each auxiliary method encloses a functionality needed to perform the queries. The three functionalities areas identified are the following ones.

- Find an element in the Dictionary: this functionality is needed to find an specific element in the dictionary, once the element has been found the position of the element in the dictionary must be obtained, because in the index the element are stored as their relative positions in the dictionary. findPosStringInDic
- Find a value in the index: this functionality is needed to find all the matches with a specific value in the index for a column selected. All the positions founded will be obtained. findPosValueInIndex
- Find all the attributes of a tuple: this functionality is needed to retrieve the values of all the attributes for a specific row. All the attributes are retrieved from index and translated with the dictionary file to get the real values. findColValueRorRows

Every query implemented in approach 03 (the most important and optimal approach implemented) is a combination of these auxiliary methods as it is described in section 4.5.2. Once the different functionalities areas are bounded, the next step is to determine how the inputs and table characteristics affect these methods.

6.2 Parameters of influence

There are three scenarios that have been widely studied along this project, linked to three queries. These queries and their characteristics are going to be briefly explained in order to introduce the second main conclusion, the impact of different parameters to each functionality area in order to be able to improve their performance in a proper way. The table used has a total of 16 columns and 6 million rows. The first query demands all the attributes of the tuples which an element specified matches with an attributed selected. (SELECT * WHERE L_COMMITDATE = "1993-08-16") The column selected has a high dictionary compression (2466 elements) and the element specified appears 2454 times. The second query demands also all the attributes of the tuples found but there are more requirements and consequently less matches found, only 15 (SELECT * WHERE L_COMMITDATE = "1993-08-16" AND L_RECEIPTDATE= "1993-10-28"). The last query extensively studied demands only 1 attribute and there was only 1 match and the column where the search is performed has a bad dictionary compression with more than 4,5 million elements. The three scenarios present different characteristics and thanks to several tests conclusions about the parameters influence has been determined.

- Find all the attributes of a tuple: the operation time needed for this functionality becomes more dominant when the numbers of tuples demanded increases, it has been tested that its operation time is principal in the first query because there are 2454 tuples requested. Also it is quite significant the number of attributes requested. It is also related to the dictionary compression of the columns of the table, due to an optimization that will be explained in next section. Another parameter that affects is the buffer size, it could be good that the size of the buffer is high in some scenarios, mainly when there are many tuples requested, then it is possible that when an IO read operation is done to translate the value of specific attribute of a tuple, and the buffer is filled the other tuples requested for that attribute have their values within the buffer already read, so the translation avoids more IO operations. But when there are not many tuples to be translated the buffer is filled and small amount of data is useful, this problem will be presented and a solution described in future works.

- Find a value in the index: this functionality is not affected by the dictionary compression not to the number of columns. It is only related to the size of the buffer, the biggest the less amount of IO operations are needed. And, of course, it is affected by the number of rows.

- Find an element in the Dictionary: this functionality was supposed to be more significant and dominant when the search is performed in a dictionary with a bad compression and the length of the elements is long. This happens in the third query presented, around 4,5 million elements, and they are long characters string because it is a column for comments. But in the experiments a powerful optimization was always running and due to this, the execution time needed for this functionality was always insignificant in all the experiments. This optimization is presented in next section.

	Buffer size	Dictionary compression	Tuples requested	Number of columns
Find an element in the Dictionary	High influence *	High influence *	Not related	Not related
Find a value in the index	High influence	Not related	Not related	Not related
Find all the values of the tuples	High influence	influence	High influence	High influence
* Before the B-tree optimization				

Figure 6.1: Functionality areas and parameters of influence

Functionality	Optimizations	Performance obtained
Find an element in the Dictionary	• B-tree	high
Find a value in the index	• Dictionary compression	Normal, slight improvement
Find all the values of the tuples	• Direct translation • Sorting • Conditional case	Normal, slight improvement

Figure 6.2: Functionality areas and optimizations

The influence of the parameters in the different functionality areas are summarized in table 6.1.

6.3 Optimizations implemented

The first and the second conclusions where related with the fragmentation and the inputs of the problem. The third main conclusion is the description of the optimizations implemented and how effective they are. In this Thesis optimizations have been implemented in the three functionalities areas described, the most part of the effort have been dedicated to the functionality ”Find an element in the Dictionary”, the optimizations implemented are described in table 6.2.

The most important optimization implemented and the one that get the highest success is the B-tree storage of the dictionary file. Thanks to it, even for the worst scenario, that should be a small buffer size, a bad compressed dictionary and long elements, the IO operation are widely reduced when performing the search. For example, in query three this worst scenario is presented and even for the smallest buffer size (4096 bytes) only 4 IO operations are needed to perform a search (16384 bytes) in a dictionary column of 4,5 million elements an average length of 35 bytes (total size 157MB). In all the queries tested the operation time of the functionality ”Find an element in the Dictionary” is not significant thanks to this optimization. When comparing approach 03 with approach 04, that not uses the optimization, the higher time differences arises in the third query due to this optimization, approach 03 is 13 second faster in average than approach 04 for this query.

In the main approach, number 03, the original dictionary compression of the table has been kept, this is not an optimization was already available due to

the table requirements. Keeping dictionary compression allows performing the search in the dictionary through int values, which means independence from the length of the elements; this will be delegate to the dictionary. Despite now two searches must be done, in the dictionary file and in the index file, performance is improved.

Another interesting optimization is the direct translation; the values in the index are not the relative positions of the elements in the dictionary, but the position of the element in the dictionary file. As each element has different length it used to require several IO operations translating the value stored in the index to the real one in the dictionary, but with this optimization many IO operations are saved, and the values can be directly translated.

Sorting it is a useful optimization that avoids many IO operations. This optimization is powerful when there are a big amount of tuples requested. When this happens and the attributes have to be recovered, this is done one by one of all the tuples requested. With the first attribute all the values stored in the index are retrieved and then the values are sorted from lower to higher, (these values are the real position of the values in the dictionary file) then an IO operation is performed in the dictionary file to translate the first element, before invoke another IO operation for the second element it is checked if the value is already available in the buffer read the first time, this easier to happen as all values are sorted and the values are the position.

The conditional case avoids sorting when the dictionary is small enough to be retrieved with only one IO operation. Then it is not useful and execution time is saved.

Chapter 7

Future works

7.1 Optimization for the B-tree creation

If the thesis is studied in detail the first optimization that should be proposed select the proper chunk size for the B-tree within the buffer size available. It has been tested that the biggest chunk is not the proper solution. When the dictionary column B-tree already have a deepness that allows a search with a small amount of IO operation, the fact of create bigger chunks due to the buffer size is higher is inefficient, the search performed in each chunk will require higher execution time.

For example if we have a B-tree of 2000 element, we can describe these two situations:

- First level 100 elements (the roots), then each root will lead to chunks of 19 elements.
- First level 1000 elements (the roots), then each root will lead to chunks of only one element.

In the first situation a buffer of 100 element is available, two IO operations are performed. After the first IO operation, a search is done through a buffer of 100 elements. And after the second IO operation the search is through 19 elements.

- Result: 2 IO operations, 119 elements reade and 119 element checked to complete the search in the worst case

In the second situation a buffer of 1000 element is available, two IO operations are performed. After the first IO operation, a search is done through a buffer of 1000 elements. And after the second IO operation the search is through 1 element.

- Result: 2 IO operations, 1001 elements read and 1001 element checked to complete the search in the worst case.

The solution implemented in this project always try to create the tree with the greater number of elements possible in the higher levels and as it has been

explained this is not always the most efficient solution. The figure 5.26 of approach 03 evaluation shows how this problem arises and when the buffer size is higher the performance is worst.

7.2 Other functionalities areas

This thesis has been quite focused in the improvement of the first step of many of the queries here studied; this is checking if the value is available in the dictionary. Good results have been obtained and the optimizations tested presents worth behavior. But the other two functionalities areas "Find a value in the index" and "Find all the values of the tuples" requires deeper study and many possibilities are available to be implemented and tested.

7.3 Updates

This thesis has not taken in to account the updates of the tables. One of the advantages introduced in the use cases was the uncommonly alteration of the tables. Due to this a complex structure such a B-tree has been used, but when modifications are done the B-tree has to be recalculated, and the index file fully updated. Due to this the best way solution should be store this occasionally updates in a log file. And when it is appropriate store the changes in dictionary file an index file.

7.4 Column Store and Row store

All the approach has been implemented following the column store format. It will be interesting to compare how all the optimizations work with the row store structure. It will be also required an study of how both structures affect the different functionality areas described.

7.5 Security

Another important issue is the security. The time required to load a full table in memory will be increased if the data has also to be decrypted. Within the solution presented in this thesis only the dictionary file should be encrypted, this encryption should be related whit the buffer size and the chunks of the trees that will be used. Therefore, only the part retrieved when performing the search in the dictionary should be decrypted.

Chapter 8

Annex

8.1 Method: generate1NtreeOrder

```
1  /*APPROACH   02 */
2  /***************/
3  //Auxiliary method
4  //This method generates the convenient order to store the data (1N)
5  vector<int> DiskLayerManager::generate1NtreeOrder(int rowsDicAux,
       int buffersizeRows){
6
7      vector<int> newSort;
8
9      //If the full dictionary is smaller than the buffer we store it
           in normal sort
10     if(rowsDicAux <= 2*buffersizeRows){
11         for(int i=0;i<rowsDicAux;i++){
12             newSort.push_back(i);
13         }
14         return newSort;
15     }
16
17     int jump = (int)( (float)rowsDicAux/(float)buffersizeRows ); //
           ceil
18     int position = 0;
19     for(int i=0;i<buffersizeRows;i++){
20         newSort.push_back(position);
21         position+=jump;
22     }
23
24     for(int i=0;i<buffersizeRows;i++){
25         int posIni = newSort.at(i);
26         int posFin = ( i==(buffersizeRows-1) ) ? rowsDicAux :
               newSort.at(i+1);
27
28         int i2 = posIni+1;
29         for(i2;i2<posFin;i2++){
30             posIni++;
31             newSort.push_back(posIni);
32         }
33     }
34
35     return newSort;
36 }
```

8.2 Method: addDataToFile_aproach03

```
1  /*APPROACH   03 */
2  /****************/
3  //Method to store the table in the disk
4  int DiskLayerManagerRE::addDataToFile_aproach03(class
       MainTableManager *MTMp, int buffersize)
5  {
6      //-------------------------//
7      //Create a DICTIONARY file
8      //-------------------------//
9      //Create a data file (content of the columns diposed in rows)
10     ofstream outDIC(filesLocation+MTMp->tableName+".dic",ios::
           binary);
11     //Create a file for index
12     ofstream outIND(filesLocation+MTMp->tableName+".index",  ios::
           binary);
13
14     int buffersizeU = buffersize - 8;//buffer size Useful (8 bytes
           reserved for number of chunks and element position)
15
16     vector<int> chunksPerCol;
17
18     //For each column
19     for(int dicCol = 0; dicCol< (MTMp->columnNum); dicCol++){
20
21          cout << " --DiCol:" << dicCol;
22
23         int currentBufferFill = 0;
24
25         int level = 0;
26         int levelaux = 0;
27         int chunkNumInLevel = 0;
28         int chunkPosition = 0;
29         int chunkPositionReal = 0;
30
31         int elementPosInLevel = 0;
32
33         string elementDic;
34
35         vector<int> columnPosition;
36         for(int i=0; i<MTMp->myColumns.at(dicCol).myDic.size() ;i
               ++){
37             columnPosition.push_back(i);
38         }
39
40         //We calculate the new order, to save later the elements in
               a B-tree way
41         newOrderMultiLevel my_noML;
42         my_noML = DiskLayerManagerRE::generateMLneworder( MTMp->
               myColumns.at(dicCol).myDic,columnPosition,buffersize,
               my_noML);
43
44         //We store the final position of the element in the file in
               a vector
45         vector<int> dicElemePosInFile;
46         dicElemePosInFile.assign(my_noML.newOrderResult.size(),0);
47
48         //We store the total number of chunks needed to save the
               dictionary of this column
49         chunksPerCol.push_back((int)my_noML.elementByChunk.size());
50
```

```
51          int numChunkReal = 0;
52          int maxInChunk = my_noML.elementByChunk.at(numChunkReal);
53          int elemenInChunk = 1;
54
55          for(int elem = 0; elem<my_noML.newOrderResult.size() ;elem
                ++){
56
57              //Check in which level of the tree we are
58              if(elemenInChunk == 1){
59                  if( level > my_noML.chunkByLevel.size() ){
60                      break;
61                  }
62                  //Number of chunks in this level
63                  if(level>levelaux){
64                      cout << " Level:" << level;
65                      levelaux = level;
66                  }
67                  chunkNumInLevel = my_noML.chunkByLevel.at(level);
68
69                  outDIC.write((char *)(&chunkNumInLevel), sizeof(
                        chunkNumInLevel));
70                  elementPosInLevel++;
71                  outDIC.write((char *)(&elementPosInLevel), sizeof(
                        elementPosInLevel));
72
73                  chunkPosition++;
74                  chunkPositionReal++;
75              }
76
77              //Get the element
78              elementDic = MTMp->myColumns.at(dicCol).myDic.at(
                    my_noML.newOrderResult.at(elem) );
79              int s_size = (int)elementDic.size();
80              currentBufferFill += s_size + 4;
81
82              if( elemenInChunk > maxInChunk ){
83
84                  //Then we have to add padding to fill the chunk so
                        we get the exact size of the buffer
85                  currentBufferFill -= ((int)elementDic.size() + 4);
86                  int padNeeded = buffersizeU - currentBufferFill;
87                  string elementPadd;
88                  elementPadd.assign(padNeeded,'\0');
89
90                  outDIC.write( elementPadd.c_str(), padNeeded );
91
92                  currentBufferFill = 0;
93                  numChunkReal++;
94                  maxInChunk = my_noML.elementByChunk.at(
                        numChunkReal);
95                  elemenInChunk = 1;
96
97                  if(chunkPositionReal == chunkNumInLevel){
98
99                      //We jump to next level
100                     chunkPosition = 0;
101                     chunkPositionReal = 0;
102                     elementPosInLevel = 0;
103                     level++;
104                     elem--;
105
106                 }else{
```

```
107
108                         chunkPosition++;
109                         elem--;
110
111                     }
112
113                 }else{
114
115                     elementPosInLevel++;
116
117                     outDIC.write((char *)(&s_size), sizeof(s_size));
118
119                     dicElemePosInFile.at( my_noML.newOrderResult.at(
                            elem) ) = ( (int)outDIC.tellp() );
120                     outDIC.write( elementDic.c_str(), elementDic.size()
                            );
121
122                     elemenInChunk++;
123                 }
124             }
125
126             //Add the padding to the last chunk of the dictionary
127             int lastPadNeeded = buffersizeU - currentBufferFill;
128             string lastPadd;
129             lastPadd.assign(lastPadNeeded,'\0');
130             outDIC.write( lastPadd.c_str(), lastPadNeeded );
131
132             //-------------------------//
133             //Create an INDEX file
134             //-------------------------//
135             for(unsigned int indIND = 0; indIND < MTMp->myColumns.at(
                    dicCol).myInd.size(); indIND++){
136                 //We store the position of the value in the file
137                 int indValue = dicElemePosInFile.at( MTMp->myColumns.
                        at(dicCol).myInd.at(indIND) );
138                 outIND.write((char *)(&indValue), sizeof(indValue));
139             }
140             //END save INDEX file
141             cout << " -- End DiCol:" << dicCol << " \n";
142         }
143
144         outIND.close();
145         outDIC.close();
146
147         //-------------------------//
148         //Create a HEADER file
149         //-------------------------//
150         //Creata header file (info about te name of the column, the max
                lentgh of their string, and number of rows)
151         ofstream outHEA(filesLocation+MTMp->tableName+".header");
152         for(int indCol = 0; indCol<MTMp->columnNum; indCol++)
153             outHEA<< MTMp->myColumns.at(indCol).columnName << "|"<<
                    chunksPerCol.at(indCol) << "|" << MTMp->myColumns.at(
                    indCol).myDic.size() << "|";
154
155         outHEA << endl << MTMp->rowsNum;
156
157         outHEA.close();
158
159         return 1;
160 }
```

8.3 Method: generateMLneworder

```
1  /*APPROACH   03 */
2  //Method to store the table in the disk
3  //Auxiliary method
4  //This method generates the convenient order to store the data (
       Multilevel mode)
5  newOrderMultiLevel DiskLayerManagerRE::generateMLneworder(vector<
       string> columnString,vector<int> columnPosition,int buffersize,
       newOrderMultiLevel my_noML){
6
7      vector<int> elementByChunkAux;
8      int numElementsLeft = (int)columnString.size();
9      vector<int> currentVectorPosStored;
10
11     vector<string> nextCurrentVector;
12     vector<int> nextCurrentVectorPos;
13
14     int buffersizeU = buffersize - 8;//buffer size Useful (8 bytes
           reserved for number of chunks and element position)
15     int currentChunkSize = 0;
16     //number of chunks in this level
17     int numOfChunks = 1;
18     //number of elements in the chunk
19     int elementsInChunk = 0;
20
21     for(int numElem = 0; numElem<numElementsLeft; numElem++){
22
23         string s_aux = columnString.at(numElem);
24
25         currentChunkSize += (int)s_aux.size() + 4; //4 bytes
               reserved to store the length
26
27         if(currentChunkSize > buffersizeU){
28
29             elementByChunkAux.push_back( elementsInChunk );
30             elementsInChunk = 0;
31
32             //We store this element for the higher level
33             nextCurrentVector.push_back(s_aux);
34             nextCurrentVectorPos.push_back( columnPosition.at(
                   numElem) );
35             //Reset currentChunkSize for next chunk
36             currentChunkSize = 0;
37
38             //1 chunk more in this level
39             if(numElem != (numElementsLeft-1) ){
40                 numOfChunks += 1;
41             }
42
43         }else{
44
45             elementsInChunk++;
46             currentVectorPosStored.push_back( columnPosition.at(
                   numElem) );
47
48         }
49     }
50     //Element in last chunk
51     elementByChunkAux.push_back( elementsInChunk );
52
53     //Add the vector calulated int this level to the final result
```

```
54      my_noML.newOrderResult.insert( my_noML.newOrderResult.begin() ,
            currentVectorPosStored.begin(),currentVectorPosStored.end
            ());
55      //Add the vector calulated "number of elments in chunk" int
            this level to the final result
56      my_noML.elementByChunk.insert( my_noML.elementByChunk.begin() ,
            elementByChunkAux.begin(),elementByChunkAux.end());
57
58      //Save the number of chunks in this level
59      my_noML.chunkByLevel.insert( my_noML.chunkByLevel.begin() ,
            numOfChunks );
60
61      if(nextCurrentVector.size()==0){
62          return my_noML;
63      }
64
65      return DiskLayerManagerRE::generateMLneworder(
            nextCurrentVector , nextCurrentVectorPos, buffersize,
            my_noML );
66  }
```

8.4 Method: findPosStringInDic

```
1   /*APP03 - Auxilary method */
2   //Find a string in a dictionary
3   int QueryManagerRE::findPosStringInDic(string elementName,int
        streamSize,HANDLE hFile,int posInFileColStart){
4
5       int posInFile = posInFileColStart;
6       //BUFFER
7       int BUFFERSIZE = streamSize;
8       DWORD dwBytesRead = 0;
9       void *ptr;
10      ptr = VirtualAlloc(NULL, BUFFERSIZE, MEM_RESERVE | MEM_COMMIT,
            PAGE_READWRITE);
11
12      //Position of the element found
13      int valuePos = 0;
14      //Bytes to jump of previous level
15      int prevLevelsChunks = 0;
16      //Num of chunks in the current level
17      int* numChunksLevel = 0;
18
19      //Result of the comparison of the strings
20      int compResult = -1;
21
22      //LOOP for chunks
23      //Go to the position where the column requested starts
24      //IF FILE_FLAG_NO_BUFFERING is selected only n*512 jumps
            available
25      while( INVALID_SET_FILE_POINTER != SetFilePointer (hFile,
            posInFile, NULL, FILE_BEGIN)){
26
27          //READ FILE
28          ReadFile(hFile, ptr, BUFFERSIZE, &dwBytesRead, NULL);
29          if(dwBytesRead == 0){
30              break;
31          }
32          numChunksLevel = (int*)ptr;
```

```
33          prevLevelsChunks += (*numChunksLevel);
34          int * positionLevel = (int *)ptr+1;
35
36          //LOOP for strings
37          int incrN = 8;
38          int incrE = 12;
39          while( incrE < (int)dwBytesRead ){
40
41              char * numCharsElemC = (char *)ptr  + incrN;
42
43              //We could have arrived to the position of the padding
44              if(numCharsElemC[0] == '\0'&& numCharsElemC[1] == '\0'
                    && numCharsElemC[2] == '\0' && numCharsElemC[3] ==
                    '\0'){
45                  //This means alphabetic higher than the last
                        element of the current chunk
46                  posInFile = posInFileColStart + prevLevelsChunks *
                        streamSize;
47                  posInFile += ((*positionLevel)-1 ) * streamSize;
48                  break;
49              }
50
51              int * numCharsElem =  (int *)numCharsElemC;
52              char* elementChars = (char*)ptr+incrE;
53              string sn = string(elementChars,*numCharsElem);
54              //We check if itÂ´s the word we are looking for
55              compResult = elementName.compare(sn);
56              if(compResult == 0){
57                  valuePos = posInFile + incrE;
58                  break;
59              }
60              if(compResult < 0){
61                  posInFile = posInFileColStart + prevLevelsChunks *
                        streamSize;
62                  posInFile += ((*positionLevel)-1 ) * streamSize;
63                  break;
64              }
65
66              incrN += (*numCharsElem) + 4;
67              incrE = incrN + 4;
68              (*positionLevel)++;
69
70              if(incrE > (int)dwBytesRead){
71                  //This means alphabetic higher than the last
                        element of the current chunk
72                  posInFile += prevLevelsChunks * streamSize;
73                  posInFile += ((*positionLevel)-1 ) * streamSize;
74                  break;
75              }
76          }
77          if(compResult == 0){
78              break;
79          }
80      }
81      CloseHandle(hFile);//CLOSE FILE
82      if(compResult == -1){
83          cout << "\n\n--VALUE NOT FOUND: " << elementName << "--\n\n
                ";
84          return -1;
85      }
86      return valuePos;
87  }
```

Bibliography

[1] Michael J. Folk, Bill Zoellick, Greg Riccardi *Instructor's Manual, File structures, An Object-Oriented Approach with C++* Chapter 3, Secondary Storage and System Software University of Illinois (document)

[2] Attendees at the Lowell Workshop were: Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. *The Lowell Database Research Self Assessment* http://research.microsoft.com/en-us/um/people/gray/lowell/LowellDatabaseResearchSelfAssessment.htm 2003. January 2013 2.1

[3] Andy Oppel *Database Demystified* page 2 Mc Graw Hill 2nd Edition, 2011. 2.2

[4] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, *Database Systems: The Complete Book* Prentice Hall, 2nd Edition, 2008. (document), 2.3, 2.1

[5] Joos-Hendrik Boese, Christian Mathis, Cafer Tosun, Franz Faerber *Data Management with SAPs In-Memory Computing Engine* http://www.edbt.org/Proceedings/2012-Berlin/papers/industrial/a4-bose.pdf 2012. January 2013 2.3.2

[6] Michael J. Folk, Bill Zoellick, Greg Riccardi *File Structures, An Object-Oriented Approach with C++* Chapter 4: Fundamental File Structure Concepts Chapter 6: Organizing Files for Performance Addison-Wesley 1998 2.4.1